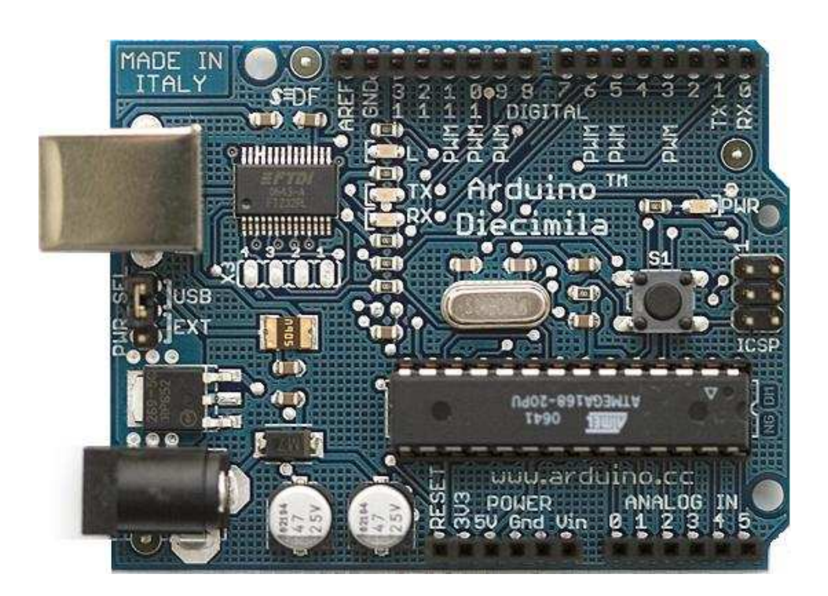
Manual de Programación

Arduino

La "inteligencia de Arduino" se expresa mediante su lenguaje de programación



Guía rápida de referencia

Traducido y adaptado:
José Manuel Ruiz Gutiérrez

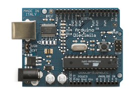
Datos del documento original

Arduino Notebook: A Beginner's Reference Written
and compiled by Brian W. Evans

With information or inspiration taken from:
http://www.arduino.cc
http://www.wiring.org.co
http://www.arduino.cc/en/Booklet/HomePage
http://cslibrary.stanford.edu/101/

Including material written by:
Massimo Banzi
Hernando Barragin
David Cuartielles
Tom Igoe
Todd Kurt
David Mellis
and others

Published:
First Edition August 2007



Índice de contenidos

estructura de un programa

La estructura básica del lenguaje de programación de Arduino es bastante simple y se compone de al menos dos partes. Estas dos partes necesarias, o funciones, encierran bloques que contienen declaraciones, estamentos o instrucciones.

```
void  setup()
{
  estamentos;
}
  void  loop()
{
  estamentos;
}
```

En donde **setup()** es la parte encargada de recoger la configuración y **loop()** es la que contienen el programa que se ejecutará cíclicamente (de ahí el termino loop –bucle-). Ambas funciones son necesarias para que el programa trabaje.

La función de configuración debe contener la declaración de las variables. Es la primera función a ejecutar en el programa, se ejecuta sólo una vez, y se utiliza para configurar o inicializar pinMode (modo de trabajo de las E/S), configuración de la comunicación en serie y otras.

La función bucle (loop) siguiente contiene el código que se ejecutara continuamente (lectura de entradas, activación de salidas, etc) Esta función es el núcleo de todos los programas de Arduino y la que realiza la mayor parte del trabajo.

setup()

La función **setup()** se invoca una sola vez cuando el programa empieza. Se utiliza para inicializar los modos de trabajo de los pins, o el puerto serie. Debe ser incluido en un programa aunque no haya declaración que ejecutar.

```
void  setup()
      {
            pinMode(pin,  OUTPUT);  // configura el  'pin'  como salida
      }
```

loop()

Después de llamar a **setup(),** la función **loop()** hace precisamente lo que sugiere su nombre, se ejecuta de forma cíclica, lo que posibilita que el programa este respondiendo continuamente ante los eventos que se produzcan en la tarjeta

```
void loop()
    {
        digitalWrite(pin, HIGH);   // pone en uno (on, 5v) el ´pin´
        delay(1000);               // espera un segundo (1000 ms)
        digitalWrite(pin, LOW);    // pone en cero (off, 0v.) el ´pin´
        delay(1000);
    }
```

funciones

Una función es un bloque de código que tiene un nombre y un conjunto de estamentos que son ejecutados cuando se llama a la función. Son funciones setup() y loop() de las que ya se ha hablado. Las funciones de usuario pueden ser escritas para realizar tareas repetitivas y para reducir el tamaño de un programa. Las funciones se declaran asociadas a un tipo de valor **"type"**. Este valor será el que devolverá la función, por ejemplo **'int'** se utilizará cuando la función devuelva un dato numérico de tipo entero. Si la función no devuelve ningún valor entonces se colocará delante la palabra "**void",** que significa "función vacía". Después de declarar el tipo de dato que devuelve la función se debe escribir el nombre de la función y entre paréntesis se escribirán, si es necesario, los parámetros que se deben pasar a la función para que se ejecute.

```
type  nombreFunción(parámetros)
    {
       estamentos;
    }
```

La función siguiente devuelve un número entero, **delayVal()** se utiliza para poner un valor de retraso en un programa que lee una variable analógica de un potenciómetro conectado a una entrada de Arduino. Al principio se declara como una variable local, ´**v**´ recoge el valor leído del potenciómetro que estará comprendido entre 0 y 1023, luego se divide el valor por 4 para ajustarlo a un margen comprendido entre 0 y 255, finalmente se devuelve el valor ´**v**´ y se retornaría al programa principal. Esta función cuando se ejecuta devuelve el valor de tipo entero ´**v**´

```
int  delayVal()
  {
     int  v;                 //  crea una variable temporal  'v'
     v= analogRead(pot);     // lee el valor del potenciómetro
     v  /=  4;               //  convierte  0-1023 a  0-255
     return  v;              //  devuelve el valor final
  }
```

{} entre llaves

Las llaves sirven para definir el principio y el final de un bloque de instrucciones. Se utilizan para los bloques de programación setup(), loop(), if.., etc.

```
type  funcion()
        {
           estamentos;
        }
```

Una llave de apertura "{" siempre debe ir seguida de una llave de cierre "}", si no es así el programa dará errores.

El entorno de programación de Arduino incluye una herramienta de gran utilidad para comprobar el total de llaves. Sólo tienes que hacer click en el punto de inserción de una llave abierta e inmediatamente se marca el correspondiente cierre de ese bloque (llave cerrada).

; punto y coma

El punto y coma ";" se utiliza para separar instrucciones en el lenguaje de programación de Arduino. También se utiliza para separar elementos en una instrucción de tipo "bucle for".

```
int  x  =  13;       //  declara la variable  'x' como tipo entero de valor  13
```

Nota: Olvidarse de poner fin a una línea con un punto y coma se traducirá en un error de compilación. El texto de error puede ser obvio, y se referirá a la falta de una coma, o puede que no. Si se produce un error raro y de difícil detección lo primero que debemos hacer es comprobar que los puntos y comas están colocados al final de las instrucciones.

/*… */ bloque de comentarios

Los bloques de comentarios, o multi-línea de comentarios, son áreas de texto ignorados por el programa que se utilizan para las descripciones del código o comentarios que ayudan a comprender el programa. Comienzan con / * y terminan con * / y pueden abarcar varias líneas.

```
/*   esto es un bloque de comentario
     no se debe olvidar cerrar los comentarios
     estos deben estar equilibrados
*/
```

Debido a que los comentarios son ignorados por el programa y no ocupan espacio en la memoria de Arduino pueden ser utilizados con generosidad y también pueden utilizarse para "comentar" bloques de código con el propósito de anotar informaciones para depuración.

Nota: Dentro de una misma línea de un bloque de comentarios no se puede escribir otra bloque de comentarios (usando /* .. */)

// línea de comentarios

Una línea de comentario empieza con / / y terminan con la siguiente línea de código. Al igual que los comentarios de bloque, los de línea son ignoradas por el programa y no ocupan espacio en la memoria.

```
// esto es un comentario
```

Una línea de comentario se utiliza a menudo después de una instrucción, para proporcionar más información acerca de lo que hace esta o para recordarla más adelante.

variables

Una variable es una manera de nombrar y almacenar un valor numérico para su uso posterior por el programa. Como su nombre indica, las variables son números que se pueden variar continuamente en contra de lo que ocurre con las constantes cuyo valor nunca cambia. Una variable debe ser declarada y, opcionalmente, asignarle un valor. El siguiente código de ejemplo declara una variable llamada *variableEntrada* y luego le asigna el valor obtenido en la entrada analógica del PIN2:

```
int variableEntrada = 0;          // declara una variable y le asigna el valor 0
variableEntrada = analogRead(2);// la variable recoge el valor analógico del PIN2
```

'variableEntrada' es la variable en sí. La primera línea declara que será de tipo entero “int”. La segunda línea fija a la variable el valor correspondiente a la entrada analógica PIN2. Esto hace que el valor de PIN2 sea accesible en otras partes del código.

Una vez que una variable ha sido asignada, o re-asignada, usted puede probar su valor para ver si cumple ciertas condiciones (instrucciones if..), o puede utilizar directamente su valor. Como ejemplo ilustrativo veamos tres operaciones útiles con variables: el siguiente código prueba si la variable *“entradaVariable”* es inferior a 100, si es cierto se asigna el valor 100 a *“entradaVariable*” y, a continuación, establece un retardo (delay) utilizando como valor “*entradaVariable”* que ahora será como mínimo de valor 100:

```
if (entradaVariable < 100) // pregunta si la variable es menor de 100
  {
    entradaVariable = 100;// si es cierto asigna el valor 100 a esta
  }
delay(entradaVariable);    // usa el valor como retardo
```

Nota: Las variables deben tomar nombres descriptivos, para hacer el código más legible. Nombres de variables pueden ser "contactoSensor" o "pulsador", para ayudar al programador y a cualquier otra persona a leer el código y entender lo que representa la variable. Nombres de variables como "var" o "valor", facilitan muy poco que el código sea inteligible. Una variable puede ser cualquier nombre o palabra que no sea una *palabra reservada* en el entorno de Arduino.

declaración de variables

Todas las variables tienen que declararse antes de que puedan ser utilizadas. Para declarar una variable se comienza por definir su tipo como **int** (entero), **long** (largo), **float** (coma flotante), etc, asignándoles siempre un nombre, y, opcionalmente, un valor inicial. Esto sólo debe hacerse una vez en un programa, pero el valor se puede cambiar en cualquier momento usando aritmética y reasignaciones diversas.

El siguiente ejemplo declara la variable *entradaVariable* como una variable de tipo entero *"int",* y asignándole un valor inicial igual a cero. Esto se llama una asignación.

```
int entradaVariable = 0;
```

Una variable puede ser declarada en una serie de lugares del programa y en función del lugar en donde se lleve a cabo la definición esto determinará en que partes del programa se podrá hacer uso de ella.

Utilización de una variable

Una variable puede ser declarada al inicio del programa antes de la parte de configuración *setup()*, a nivel local dentro de las funciones, y, a veces, dentro de un bloque, como para los bucles del tipo if.. for.., etc. En función del lugar de declaración de la variable así se determinara el ámbito de aplicación, o la capacidad de ciertas partes de un programa para hacer uso de ella.

Una *variable global* es aquella que puede ser vista y utilizada por cualquier función y estamento de un programa. Esta variable se declara al comienzo del programa, antes de *setup()*.

Una *variable local* es aquella que se define dentro de una función o como parte de un bucle. Sólo es visible y sólo puede utilizarse dentro de la función en la que se declaró.

Por lo tanto, es posible tener dos o más variables del mismo nombre en diferentes partes del mismo programa que pueden contener valores diferentes. La garantía de que sólo una función tiene acceso a sus variables dentro del programa simplifica y reduce el potencial de errores de programación.

El siguiente ejemplo muestra cómo declarar a unos tipos diferentes de variables y la visibilidad de cada variable:

```
int value; // 'value' es visible para cualquier función
void setup()
      {
            // no es necesario configurar
      }

void loop()
      {
         for (int i=0; i<20;)      // 'i' solo es visible
             {    // dentro del bucle for
                  i++;
             }
         float f; // 'f' es visible solo
      }     // dentro del bucle
```

byte

Byte almacena un valor numérico de 8 bits sin decimales. Tienen un rango entre 0 y 255

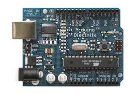
```
byte unaVariable = 180; // declara 'unaVariable' como tipo byte
```



Enteros son un tipo de datos primarios que almacenan valores numéricos de 16 bits sin decimales comprendidos en el rango 32,767 to -32,768.

```
int unaVariable = 1500; // declara 'unaVariable' como una variable
                        de tipo entero
```

Nota: Las variables de tipo entero "int" pueden sobrepasar su valor máximo o mínimo como consecuencia de una operación. Por ejemplo, si x = 32767 y una posterior declaración agrega 1 a x, x = x + 1 entonces el valor se x pasará a ser -32.768. (algo así como que el valor da la vuelta)





long

El formato de variable numérica de tipo extendido “long” se refiere a números enteros (tipo 32 bits) sin decimales que se encuentran dentro del rango -2147483648 a 2147483647.

long unaVariable = 90000; *// declara 'unaVariable' como tipo long*



float

El formato de dato del tipo “punto flotante” “float” se aplica a los números con decimales. Los números de punto flotante tienen una mayor resolución que los de 32 bits con un rango comprendido 3.4028235E +38 a +38-3.4028235E.

float unaVariable = 3.14*;* *// declara 'unaVariable' como tipo flotante*

Nota: Los números de punto flotante no son exactos, y pueden producir resultados extraños en las comparaciones. Los cálculos matemáticos de punto flotante son también mucho más lentos que los del tipo de números enteros, por lo que debe evitarse su uso si es posible.

arrays

Un array es un conjunto de valores a los que se accede con un número índice. Cualquier valor puede ser recogido haciendo uso del nombre de la matriz y el número del índice. El primer valor de la matriz es el que está indicado con el índice 0, es decir el primer valor del conjunto es el de la posición 0. Un array tiene que ser declarado y opcionalmente asignados valores a cada posición antes de ser utilizado

```
int miArray[] = {valor0, valor1, valor2...}
```

Del mismo modo es posible declarar una matriz indicando el tipo de datos y el tamaño y posteriormente, asignar valores a una posición especifica:

```
int miArray[5];        // declara un array de enteros de 6 posiciones
miArray[3] = 10;       // asigna l valor 10 a la posición 4
```

Para leer de un array basta con escribir el nombre y la posición a leer:

```
x = miArray[3];        // x ahora es igual a 10 que está en la posición 3
                       del array
```

Las matrices se utilizan a menudo para estamentos de tipo bucle, en los que la variable de incremento del contador del bucle se utiliza como índice o puntero del array. El siguiente ejemplo usa una matriz para el parpadeo de un LED.

Utilizando un bucle tipo *for*, el contador comienza en cero 0 y escribe el valor que figura en la posición de índice 0 en la serie que hemos escrito dentro del array parpadeo[], en este caso 180, que se envía a la salida analógica tipo PWM configurada en el PIN10, se hace una pausa de 200 ms y a continuación se pasa al siguiente valor que asigna el índice "i".

```
int ledPin = 10;        // Salida LED en el PIN 10
byte parpadeo[] = {180, 30, 255, 200, 10, 90, 150, 60}; // array de 8 valores
                                                           diferentes
    void setup()
    {
      pinMode(ledPin, OUTPUT); //configura la salida PIN 10
    }

      void loop()        // bucle del programa

    {
      for(int i=0; i<8; i++) // crea un bucle tipo for utilizando la variable i de 0 a 7
    {

      analogWrite(ledPin, parpadeo[i]); // escribe en la salida PIN 10 el valor al
                                           que apunta i dentro del array
                                           parpadeo[]
    delay(200);   // espera 200ms

    }
}
```

aritmética

Los operadores aritméticos que se incluyen en el entorno de programación son suma, resta, multiplicación y división. Estos devuelven la suma, diferencia, producto, o cociente (respectivamente) de dos operandos

```
y = y + 3;
x = x - 7;
i = j * 6;
r = r / 5;
```

La operaciones se efectúa teniendo en cuanta el tipo de datos que hemos definido para los operandos (int, dbl, float, etc..), por lo que, por ejemplo, si definimos 9 y 4 como enteros "int", 9 / 4 devuelve de resultado 2 en lugar de 2,25 ya que el 9 y 4 se valores de tipo entero "int" (enteros) y no se reconocen los decimales con este tipo de datos.

Esto también significa que la operación puede sufrir un desbordamiento si el resultado es más grande que lo que puede ser almacenada en el tipo de datos. Recordemos el alcance de los tipos de datos numéricos que ya hemos explicado anteriormente.

Si los operandos son de diferentes tipos, para el cálculo se utilizará el tipo más grande de los operandos en juego. Por ejemplo, si uno de los números (operandos) es del tipo float y otra de tipo integer, para el cálculo se utilizará el método de float es decir el método de coma flotante.

Elija el tamaño de las variables de tal manera que sea lo suficientemente grande como para que los resultados sean lo precisos que usted desea. Para las operaciones que requieran decimales utilice variables tipo float, pero sea consciente de que las operaciones con este tipo de variables son más lentas a la hora de realizarse el computo..

Nota: Utilice el operador *(int) myFloat* para convertir un tipo de variable a otro sobre la marcha. Por ejemplo, **i = (int) 3,6** establecerá i igual a **3**.

asignaciones compuestas

Las asignaciones compuestas combinan una operación aritmética con una variable asignada. Estas son comúnmente utilizadas en los bucles tal como se describe más adelante. Estas asignaciones compuestas pueden ser:

```
x ++     // igual que x = x + 1,   o incrementar x en + 1
x --     // igual que x = x - 1,   o decrementar x en -1
x += y   // igual que x = x + y,   o incrementra x en +y
x -= y   // igual que x = x - y,   o decrementar x en -y
x *= y   // igual que x = x * y,   o multiplicar x por y
x /= y   // igual que x = x / y,   o dividir x por y
```

Nota: Por ejemplo, x * = 3 hace que x se convierta en el triple del antiguo valor x y por lo tanto x es reasignada al nuevo valor .

operadores de comparación

Las comparaciones de una variable o constante con otra se utilizan con frecuencia en las estructuras condicionales del tipo *if*.. para testear si una condición es verdadera. En los ejemplos que siguen en las próximas páginas se verá su utilización práctica usando los siguientes tipo de condicionales:

```
x == y   // x es igual a y
x != y   // x no es igual a y
x <  y   // x es menor que y
x >  y   // x es mayor que y
```

```
x <= y     // x es menor o igual que  y
x >= y     // x es mayor o igual que y
```

operadores lógicos

Los operadores lógicos son usualmente una forma de comparar dos expresiones y devolver un **VERDADERO** o **FALSO** dependiendo del operador. Existen tres operadores lógicos, **AND (&&)**, **OR (||)** y **NOT (!)**, que a menudo se utilizan en estamentos de tipo if..:

```
Logical AND:
if (x > 0 && x < 5)      // cierto sólo si las dos expresiones son ciertas

Logical OR:
if (x > 0 || y > 0)      // cierto si una cualquiera de las expresiones
                         es cierta

Logical NOT:
if (!x > 0)              //  cierto solo si la expresión es falsa
```

constantes

El lenguaje de programación de Arduino tiene unos valores predeterminados, que son llamados constantes. Se utilizan para hacer los programas más fáciles de leer. Las constantes se clasifican en grupos.

cierto/falso (true/false)

Estas son constantes booleanas que definen los niveles **HIGH** (alto) y **LOW** (bajo) cuando estos se refieren al estado de las salidas digitales. **FALSE** se asocia con **0** (cero), mientras que **TRUE** se asocia con **1**, pero TRUE también puede ser cualquier otra cosa excepto cero. Por lo tanto, en sentido booleano, -1, 2 y -200 son todos también se define como TRUE. (esto es importante tenerlo en cuanta)

```
if (b == TRUE);
{
ejecutar las instrucciones;
}
```

high/low

Estas constantes definen los niveles de salida altos o bajos y se utilizan para la lectura o la escritura digital para las patillas. **ALTO** se define como en la lógica de nivel 1, **ON**, ó 5 voltios, mientras que **BAJO** es lógica nivel 0, **OFF**, o 0 voltios.

```
digitalWrite(13,  HIGH); // activa la salida 13 con un nivel alto (5v.)
```

input/output

Estas constantes son utilizadas para definir, al comienzo del programa, el modo de funcionamiento de los pines mediante la instrucción *pinMode* de tal manera que el pin puede ser una **entrada INPUT** o una **salida OUTPUT**.

```
pinMode(13,  OUTPUT); // designamos que el PIN 13 es una salida
```

if (si)

if es un estamento que se utiliza para probar si una determinada condición se ha alcanzado, como por ejemplo averiguar si un valor analógico está por encima de un cierto número, y ejecutar una serie de declaraciones (operaciones) que se escriben dentro de llaves, si es verdad. Si es falso (la condición no se cumple) el programa salta y no ejecuta las operaciones que están dentro de las llaves, El formato para if es el siguiente:

```
if (unaVariable ?? valor)
{
ejecutaInstrucciones;
}
```

En el ejemplo anterior se compara una variable con un valor, el cual puede ser una variable o constante. Si la comparación, o la condición entre paréntesis se cumple (es cierta), las declaraciones dentro de los corchetes se ejecutan. Si no es así, el programa salta sobre ellas y sigue.

Nota: Tenga en cuenta el uso especial del símbolo '=', poner dentro de if (x = 10), podría parecer que es valido pero sin embargo no lo es ya que esa expresión asigna el valor 10 a la variable x, por eso dentro de la estructura if se utilizaría X==10 que en este caso lo que hace el programa es comprobar si el valor de x es 10.. Ambas cosas son distintas por lo tanto dentro de las estructuras if, cuando se pregunte por un valor se debe poner el signo doble de igual “==”

if… else (si….. sino ..)

if… else viene a ser un estructura que se ejecuta en respuesta a la *idea "si esto no se cumple haz esto otro"*. Por ejemplo, si se desea probar una entrada digital, y hacer una cosa si la entrada fue alto o hacer otra cosa si la entrada es baja, usted escribiría que de esta manera:

```
if (inputPin == HIGH) // si el valor de la entrada inputPin es alto
    {
        instruccionesA; //ejecuta si se cumple la condición
    }
else
    {
        instruccionesB; //ejecuta si no se cumple la condición
    }
```

Else puede ir precedido de otra condición de manera que se pueden establecer varias estructuras condicionales de tipo unas dentro de las otras (anidamiento) de forma que sean mutuamente excluyentes pudiéndose ejecutar a la vez. Es incluso posible tener un número ilimitado de estos condicionales. Recuerde sin embargo qué sólo un conjunto de declaraciones se llevará a cabo dependiendo de la condición probada:

```
if (inputPin < 500)
    {
        instruccionesA; // ejecuta las operaciones A
    }
else if (inputPin >= 1000)
    {
        instruccionesB; // ejecuta las operacione B
    }
else
    {
        instruccionesC; // ejecuta las operaciones C
    }
```

Nota: Un estamento de tipo if prueba simplemente si la condición dentro del paréntesis es verdadera o falsa. Esta declaración puede ser cualquier declaración válida. En el anterior ejemplo, si cambiamos y ponemos (inputPin == HIGH). En este caso, el estamento if sólo chequearía si la entrada especificado esta en nivel alto (HIGH), o +5 v.

for

La declaración **for** se usa para repetir un bloque de sentencias encerradas entre llaves un número determinado de veces. Cada vez que se ejecutan las instrucciones del bucle se

vuelve a testear la condición. La declaración for tiene tres partes separadas por (;) vemos el ejemplo de su sintaxis:

```
for (inicialización; condición; expresión)
{
    ejecutaInstrucciones;
}
```

La *inicialización* de una variable local se produce una sola vez y la *condición* se testea cada vez que se termina la ejecución de las instrucciones dentro del bucle. Si la condición sigue cumpliéndose, las instrucciones del bucle se vuelven a ejecutar. Cuando la condición no se cumple, el bucle termina.

El siguiente ejemplo inicia el entero i en el 0, y la condición es probar que el valor es inferior a 20 y si es cierto i se incrementa en 1 y se vuelven a ejecutar las instrucciones que hay dentro de las llaves:

```
for (int i=0; i<20; i++)        // declara i, prueba que es menor que
                                20, incrementa i en 1
{
    digitalWrite(13, HIGH);     // envía un 1 al pin 13
    delay(250);                 // espera ¼ seg.
    digitalWrite(13, LOW);      // envía un 0 al pin 13
    delay(250);                 // espera ¼ de seg.
}
```

Nota: El bucle en el lenguaje C es mucho más flexible que otros bucles encontrados en algunos otros lenguajes de programación, incluyendo BASIC. Cualquiera de los tres elementos de cabecera puede omitirse, aunque el punto y coma es obligatorio. También las declaraciones de inicialización, condición y expresión puede ser cualquier estamento válido en lenguaje C sin relación con las variables declaradas. Estos tipos de estados son raros pero permiten disponer soluciones a algunos problemas de programación raras.

while

Un bucle del tipo **while** es un bucle de ejecución continua mientras se cumpla la expresión colocada entre paréntesis en la cabecera del bucle. La *variable de prueba* tendrá que cambiar para salir del bucle. La situación podrá cambiar a expensas de una expresión dentro el código del bucle o también por el cambio de un valor en una entrada de un sensor

```
while (unaVariable ?? valor)
{
```

```
        ejecutarSentencias;
}
```

El siguiente ejemplo testea si la variable "unaVariable" es inferior a 200 y, si es verdad, ejecuta las declaraciones dentro de los corchetes y continuará ejecutando el bucle hasta que 'unaVariable' no sea inferior a 200.

```
While (unaVariable < 200)      // testea si es menor que 200
{
        instrucciones;         // ejecuta las instrucciones entre llaves
        unaVariable++;         // incrementa la variable en 1
}
```

do… while

El bucle **do while** funciona de la misma manera que el bucle while, con la salvedad de que la condición se prueba al final del bucle, por lo que el bucle siempre se ejecutará al menos una vez.

```
do
{
        Instrucciones;
} while (unaVariable ?? valor);
```

El siguiente ejemplo asigna el valor leído *leeSensor()* a la variable 'x', espera 50 milisegundos, y luego continua mientras que el valor de la 'x' sea inferior a 100:

```
do
{
        x = leeSensor();
        delay(50);
} while (x < 100);
```

pinMode(pin, mode)

Esta instrucción es utilizada en la parte de configuración *setup ()* y sirve para configurar el modo de trabajo de un **PIN** pudiendo ser **INPUT** (entrada) u **OUTPUT** (salida).

```
pinMode(pin, OUTPUT);  // configura 'pin' como salida
```

Los terminales de Arduino, por defecto, están configurados como entradas, por lo tanto no es necesario definirlos en el caso de que vayan a trabajar como entradas. Los pines

configurados como entrada quedan, bajo el punto de vista eléctrico, como entradas en estado de alta impedancia.

Estos pines tienen a nivel interno una resistencia de 20 KΩ a las que se puede acceder mediante software. Estas resistencias se accede de la siguiente manera:

```
pinMode(pin, INPUT);      // configura el 'pin' como entrada
digitalWrite(pin, HIGH);  // activa las resistencias internas
```

Las resistencias internas normalmente se utilizan para conectar las entradas a interruptores. En el ejemplo anterior no se trata de convertir un pin en salida, es simplemente un método para activar las resistencias interiores.

Los pins configurado como OUTPUT (salida) se dice que están en un estado de baja impedancia estado y pueden proporcionar **40 mA** (miliamperios) de corriente a otros dispositivos y circuitos. Esta corriente es suficiente para alimentar un diodo LED (no olvidando poner una resistencia en serie), pero no es lo suficiente grande como para alimentar cargas de mayor consumo como relés, solenoides, o motores.

Un cortocircuito en las patillas Arduino provocará una corriente elevada que puede dañar o destruir el chip Atmega. A menudo es una buena idea conectar en la OUTUPT (salida) una resistencia externa de 470 o de 1000 Ω.

digitalRead(pin)

Lee el valor de un pin (definido como digital) dando un resultado **HIGH** (alto) o **LOW** (bajo). El pin se puede especificar ya sea como una variable o una constante (0-13).

```
valor = digitalRead(Pin);  // hace que 'valor sea igual al estado leído
                           en ´Pin´
```

digitalWrite(pin, value)

Envía al ´pin´ definido previamente como OUTPUT el valor HIGH o LOW (poniendo en 1 o 0 la salida). El pin se puede especificar ya sea como una variable o como una constante (0-13).

```
digitalWrite(pin, HIGH);  // deposita en el 'pin' un valor HIGH (alto o 1)
```

El siguiente ejemplo lee el estado de un pulsador conectado a una entrada digital y lo escribe en el ´pin´de salida LED:

```
int led      = 13;     // asigna a LED el valor 13
int boton  =     7;     // asigna a botón el valor  7
int valor =      0;     // define el valor y le asigna el valor 0

void setup()
{
      pinMode(led, OUTPUT); // configura el led (pin13) como salida
      pinMode(boton, INPUT); // configura botón (pin7) como entrada
}

void loop()
{
      valor = digitalRead(boton); //lee el estado de la entrada botón
      digitalWrite(led, valor); // envía a la salida ´led´el valor leído
}
```

analogRead(pin)

Lee el valor de un determinado pin definido como entrada analógica con una *resolución de 10 bits*. Esta instrucción sólo funciona en los pines (0-5). El rango de valor que podemos leer oscila de 0 a 1023.

```
valor = analogRead(pin);  // asigna a valor  lo que lee en la entrada ´pin'
```

Nota: Los pins analógicos (0-5) a diferencia de los pines digitales, no necesitan ser declarados como INPUT u OUPUT ya que son siempre INPUT´s.

analogWrite(pin, value)

Esta instrucción sirve para escribir un pseudo-valor analógico utilizando el procedimiento de modulación por ancho de pulso (PWM) a uno de los pin´s de Arduino marcados como *"pin PWM"*. El más reciente Arduino, que implementa el chip **ATmega168, permite habilitar como salidas analógicas tipo PWM los pines 3, 5, 6, 9, 10 y 11**. Los modelos de Arduino más antiguos que implementan el chip **ATmega8, solo tiene habilitadas para esta función los pines 9, 10 y 11**. El valor que se puede enviar a estos pines de salida analógica puede darse en forma de variable o constante, pero siempre con un margen de 0-255.

```
analogWrite(pin, valor);    // escribe 'valor' en el 'pin' definido como
                            analógico
```

Si enviamos el valor 0 genera una salida de 0 voltios en el pin especificado; un valor de 255 genera una salida de 5 voltios de salida en el pin especificado. Para valores de entre 0 y 255, el pin saca tensiones entre 0 y 5 voltios - el valor HIGH de salida equivale a 5v (5 voltios). Teniendo en cuenta el concepto de señal PWM , por ejemplo, un valor de 64

equivaldrá a mantener 0 voltios de tres cuartas partes del tiempo y 5 voltios a una cuarta parte del tiempo; un valor de 128 equivaldrá a mantener la salida en 0 la mitad del tiempo y 5 voltios la otra mitad del tiempo, y un valor de 192 equivaldrá a mantener en la salida 0 voltios una cuarta parte del tiempo y de 5 voltios de tres cuartas partes del tiempo restante.

Debido a que esta es una función de hardware, en el pin de salida analógica (PWN) se generará una onda constante después de ejecutada la instrucción *analogWrite* hasta que se llegue a ejecutar otra instrucción *analogWrite* (o una llamada a digitalRead o digitalWrite en el mismo pin).

Nota: Las salidas analógicas a diferencia de las digitales, no necesitan ser declaradas como INPUT u OUTPUT..

El siguiente ejemplo lee un valor analógico de un pin de entrada analógica, convierte el valor dividiéndolo por 4, y envía el nuevo valor convertido a una salida del tipo PWM o salida analógica:

```
int led = 10;           // define el pin 10 como ´led´
int analog =    0;      // define el pin 0 como ´analog´
int valor;              // define la variable ´valor´

void setup(){}          // no es necesario configurar entradas y salidas

void loop()
{
        valor = analogRead(analog);     // lee el pin 0 y lo asocia a la
                                        variable valor
        valor /= 4; / /divide valor entre 4 y lo reasigna a valor
        analogWrite(led, value); // escribe en el pin10 valor
}
```

delay(ms)

Detiene la ejecución del programa la cantidad de tiempo en ms que se indica en la propia instrucción. De tal manera que 1000 equivale a 1seg.

```
delay(1000);      // espera 1 segundo
```

millis()

Devuelve el número de milisegundos transcurrido desde el inicio del programa en Arduino hasta el momento actual. Normalmente será un valor grande (dependiendo del

tiempo que este en marcha la aplicación después de cargada o después de la última vez que se pulsó el botón “reset” de la tarjeta)..

valor = millis(); *// valor recoge el número de milisegundos*

Nota: Este número se desbordará (si no se resetea de nuevo a cero), después de aproximadamente 9 horas.

min(x, y)

Calcula el mínimo de dos números para cualquier tipo de datos devolviendo el número más pequeño.

valor = min(valor, 100); *// asigna a valor el más pequeños de los dos números especificados.*

Si 'valor' es menor que 100 valor recogerá su propio valor si ´valor´ es mayor que 100 valor pasara a valer 100.

max(x, y)

Calcula el máximo de dos números para cualquier tipo de datos devolviendo el número mayor de los dos.

valor = max(valor, 100); *// asigna a valor el mayor de los dos números 'valor' y 100.*

De esta manera nos aseguramos de que valor será como mínimo 100.

randomSeed(seed)

Establece un valor, o semilla, como punto de partida para la función *random().*

randomSeed(valor); *// hace que valor sea la semilla del random*

Debido a que Arduino es incapaz de crear un verdadero número aleatorio, *randomSeed* le permite colocar una variable, constante, u otra función de control dentro de la función random, lo que permite generar números aleatorios "al azar". Hay una variedad de semillas, o funciones, que pueden ser utilizados en esta función, incluido

millis () o incluso analogRead () que permite leer ruido eléctrico a través de un pin analógico.

random(max)
random(min, max)

La función **random** devuelve un número aleatorio entero de un intervalo de valores especificado entre los valores min y max.

```
valor = random(100, 200);     // asigna a la variable 'valor' un numero aleatorio
                              comprendido entre 100-200
```

Nota: Use esta función después de usar el randomSeed().

El siguiente ejemplo genera un valor aleatorio entre 0-255 y lo envía a una salida analógica PWM :

```
int randNumber;         // variable que almacena el valor aleatorio
int led = 10;           // define led como 10

void setup() {}         // no es necesario configurar nada

void loop()
{
randomSeed(millis());            // genera una semilla para aleatorio a partir
                                 de la función millis()
randNumber = random(255);        // genera número aleatorio entre 0-255
analogWrite(led, randNumber);    // envía a la salida led de tipo PWM el valor
delay(500);                      // espera 0,5 seg.
}
```

Serial.begin(rate)

Abre el puerto serie y fija la velocidad en baudios para la transmisión de datos en serie. El valor típico de velocidad para comunicarse con el ordenador es 9600, aunque otras velocidades pueden ser soportadas.

```
void setup()
    {
        Serial.begin(9600);     // abre el Puerto serie
    }                           // configurando la velocidad en 9600 bps
```

Nota: Cuando se utiliza la comunicación serie los pins digital 0 (RX) y 1 (TX) no puede utilizarse al mismo tiempo.

Serial.println(data)

Imprime los datos en el puerto serie, seguido por un retorno de carro automático y salto de línea. Este comando toma la misma forma que Serial.print (), pero es más fácil para la lectura de los datos en el Monitor Serie del software.

```
Serial.println(analogValue);        // envía el valor  'analogValue' al puerto
```

Nota: Para obtener más información sobre las distintas posibilidades de Serial.println () y Serial.print () puede consultarse el sitio web de Arduino.

El siguiente ejemplo toma de una lectura analógica pin0 y envía estos datos al ordenador cada 1 segundo.

```
void  setup()
      {
            Serial.begin(9600);     // configura el puerto serie  a   9600bps
      }

void  loop()
      {
            Serial.println(analogRead(0));  //  envía valor analógico
            delay(1000);                    //  espera 1 segundo
      }
```

Serial.printnl(data, data type)

Vuelca o envía un número o una cadena de carateres al puerto serie, seguido de un caracter de retorno de carro "CR" (ASCII 13, or '\r')y un caracter de salto de línea "LF"(ASCII 10, or '\n'). Toma la misma forma que el comando Serial.print()

Serial.println(b) vuelca o envía el valor de b como un número decimal en caracteres ASCII seguido de "CR" y "LF".

Serial.println(b, DEC) vuelca o envía el valor de b como un número decimal en caracteres ASCII seguido de "CR" y "LF".

Serial.println(b, HEX) vuelca o envía el valor de b como un número hexdecimal en caracteres ASCII seguido de "CR" y "LF".

Serial.println(b, OCT) vuelca o envía el valor de b como un número Octal en caracteres ASCII seguido de "CR" y "LF".

Serial.println(b, BIN) vuelca o envía el valor de b como un número binario en caracteres ASCII seguido de "CR" y "LF".

Serial.print(b, BYTE) vuelca o envía el valor de b como un byteseguido de "CR" y "LF".

Serial.println(str) vuelca o envía la cadena de caracteres como una cadena ASCII seguido de "CR" y "LF".

Serial.println() sólo vuelca o envía "CR" y "LF". Equivaldría a printNewline().

Serial.print(data, data type)

Vuelca o envía un número o una cadena de carateres, al puerto serie. Dicho comando puede tomar diferentes formas, dependiendo de los parámetros que utilicemos para definir el formato de volcado de los números.

Parámetros

data: el número o la cadena de caracteres a volcar o enviar.

data type: determina el formato de salida de los valores numéricos (decimal, octal, binario, etc...) **DEC, OCT, BIN, HEX, BYTE , si no se pe nada vuelva ASCII**

Ejemplos

Serial.print(b)

Vuelca o envía el valor de b como un número decimal en caracteres ASCII. Equivaldría a printInteger().

int b = 79; Serial.print(b); // *prints the string "79".*

Serial.print(b, DEC)

Vuelca o envía el valor de b como un número decimal en caracteres ASCII.

Equivaldría a printInteger().

int b = 79;
Serial.print(b, DEC); // *prints the string "79".*

Serial.print(b, HEX)

Vuelca o envía el valor de b como un número hexdecimal en caracteres ASCII. Equivaldría a printHex();

int b = 79;
Serial.print(b, HEX); // *prints the string "4F".*

Serial.print(b, OCT)

Vuelca o envía el valor de b como un número Octal en caracteres ASCII. Equivaldría a printOctal();

int b = 79;
Serial.print(b, OCT); *// prints the string "117".*

Serial.print(b, BIN)

Vuelca o envía el valor de b como un número binario en caracteres ASCII. Equivaldría a printBinary();

int b = 79;
Serial.print(b, BIN); *// prints the string "1001111".*

Serial.print(b, BYTE)

Vuelca o envía el valor de b como un byte. Equivaldría a printByte();

int b = 79;
Serial.print(b, BYTE); // *Devuelve el caracter "O", el cual representa el caracter ASCII del valor 79. (Ver tabla ASCII).*

Serial.print(str)

Vuelca o envía la cadena de caracteres como una cadena ASCII. Equivaldría a printString().

Serial.print("Hello World!"); *// vuelca "Hello World!".*

Serial.avaible()

int Serial.available()

Obtiene un número entero con el número de bytes (caracteres) diponibles para leer o capturar desde el puerto serie. Equivaldría a la función serialAvailable().

Devuelve Un entero con el número de bytes disponibles para leer desde el buffer serie, o 0 si no hay ninguno. Si hay algún dato disponible, SerialAvailable() será mayor que 0. El buffer serie puede almacenar como máximo 64 bytes.

Ejemplo

```
int incomingByte = 0; // almacena el dato serie
void setup() {
    Serial.begin(9600); // abre el puerto serie, y le asigna la velocidad de 9600
bps
}
void loop() {
    // envía datos sólo si los recibe:
    if (Serial.available() > 0) {
            // lee el byte de entrada:
            incomingByte = Serial.read();
        //lo vuelca a pantalla
            Serial.print("I received: ");
            Serial.println(incomingByte, DEC);
    }
}
```

Serial.Read()

int Serial.Read()

Lee o captura un byte (un caracter) desde el puerto serie. Equivaldría a la función serialRead().

Devuelve :El siguiente byte (carácter) desde el puerto serie, o -1 si no hay ninguno.

Ejemplo

```
int incomingByte = 0; // almacenar el dato serie
void setup() {
    Serial.begin(9600); // abre el puerto serie,y le asigna la velocidad de 9600
bps
}
void loop() {
    // envía datos sólo si los recibe:
    if (Serial.available() > 0) {
            // lee el byte de entrada:
            incomingByte = Serial.read();
        //lo vuelca a pantalla
            Serial.print("I received: ");
            Serial.println(incomingByte, DEC);
    }
}
```

Apendices

Formas de Conexionado de entradas y salidas

salida digital



Éste es el ejemplo básico equivalente al "hola mundo" de cualquier lenguaje de programación haciendo simplemente el encendido y apagado de un led. En este ejemplo el LED está conectado en el pin13, y se enciende y apaga “parpadea” cada segundo. La resistencia que se debe colocar en serie con el led en este caso puede omitirse ya que el pin13 de Arduino ya incluye en la tarjeta esta resistencia,

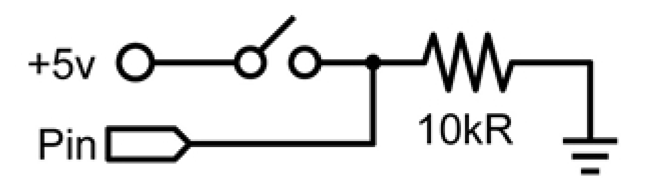
```
int ledPin = 13; // LED en el pin digital 13

void setup()      // configura el pin de salida

{
        pinMode(ledPin, OUTPUT); // configura el pin 13 como
salida
}
        void loop()      // inicia el bucle del programa
{
        digitalWrite(ledPin, HIGH); // activa el LED
        delay(1000); // espera 1 segundo
        digitalWrite(ledPin, LOW); // desactiva el LED

        delay(1000); // espera 1 segundo
}
```

entrada digital



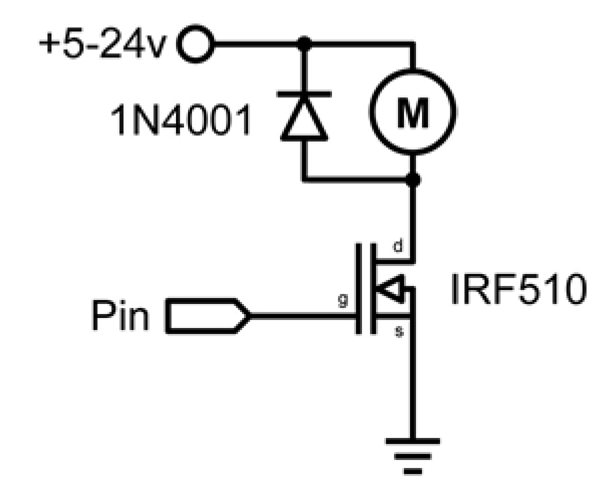
Ésta es la forma más sencilla de entrada con sólo dos posibles estados: encendido o apagado. En este ejemplo se lee un simple switch o pulsador conectado a PIN2. Cuando el interruptor está cerrado el pin de entrada se lee ALTO y encenderá un LED colocado en el PIN13

```
int  ledPin  =  13; //  pin 13 asignado para el LED de salida
int  inPin  =  2; //  pin 2 asignado para el pulsador

void  setup() // Configura entradas y salidas
	{
		pinMode(ledPin,  OUTPUT); //  declara  LED  como salida
		pinMode(inPin,  INPUT); //  declara  pulsador como entrada
	}

void  loop()
	{
		if  (digitalRead(inPin)  ==  HIGH)  //  testea si la entrada esta activa HIGH
	{
		digitalWrite(ledPin,  HIGH); //  enciende el LED
		delay(1000); //  espera 1 segundo
		digitalWrite(ledPin,  LOW); //  apaga el LED
		}
	}
```

salida de alta corriente de consumo



A veces es necesario controlar cargas de más de los 40 mA que es capaz de suministrar la tarjeta Arduino. En este caso se hace uso de un transistor MOSFET que puede alimentar cargas de mayor consumo de corriente. El siguiente ejemplo muestra como el transistor MOSFET conmuta 5 veces cada segundo.

Nota: El esquema muestra un motor con un diodo de protección por ser una carga inductiva. En los casos que las cargas no sean inductivas no será necesario colocar el diodo.

```
int  outPin = 5;   // pin de salida para el MOSFET

void  setup()
      {
            pinMode(outPin, OUTPUT);  // pin5 como salida
      }

void  loop()
      {
            for  (int  i=0;  i<=5;  i++)     //  repetir bucle 5 veces
            {
            digitalWrite(outPin,  HIGH); //  activa el MOSFET
            delay(250); //  espera 1/4  segundo
            digitalWrite(outPin,  LOW); //  desactiva el MOSFET
            delay(250); // espera 1/4  segundo
            }
            delay(1000); //  espera 1 segundo

      }
```

salida analógica del tipo pwm
PWM (modulación de impulsos en frecuncia)



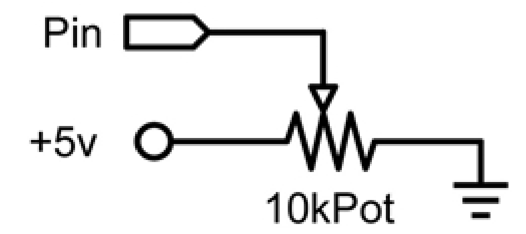
La Modulación de Impulsos en Frecuencia (PWM) es una forma de conseguir una "falsa" salida analógica. Esto podría ser utilizado para modificar el brillo de un LED o controlar un servo motor. El siguiente ejemplo lentamente hace que el LED se ilumine y se apague haciendo uso de dos bucles.

```
int ledPin = 9;  // pin PWM para el  LED

void setup(){} // no es necesario configurar nada

void loop()
  {
     for (int i=0; i<=255; i++) // el valor de i asciende
     {
          analogWrite(ledPin, i); // se escribe el valor de I en el PIN de salida del LED
          delay(100); // pauses for 100ms
     }
     for (int i=255; i>=0; i--) // el valor de I desciendei
     {
          analogWrite(ledPin, i); // se escribe el valor de ii
          delay(100); // pasusa durante 100ms
     }
  }
```

entrada con potenciómetro
(entrada analógica)



El uso de un potenciómetro y uno de los pines de entrada analógica-digital de Arduino (ADC) permite leer valores analógicos que se convertirán en valores dentro del rango de 0-1024. El siguiente ejemplo utiliza un potenciómetro para controlar un el tiempo de parpadeo de un LED.

```
int potPin = 0; // pin entrada para potenciómetro
int ledPin = 13;      // pin de salida para el LED

void setup()
      {
            pinMode(ledPin, OUTPUT);      // declara ledPin como SALIDA
      }

void loop()
      {
            digitalWrite(ledPin, HIGH); // pone ledPin en on
            delay(analogRead(potPin)); // detiene la ejecución un tiempo "potPin"
            digitalWrite(ledPin, LOW); // pone ledPin en off
            delay(analogRead(potPin)); // detiene la ejecución un tiempo "potPin"

}
```

entrada conectada a resistencia variable
(entrada analógica)



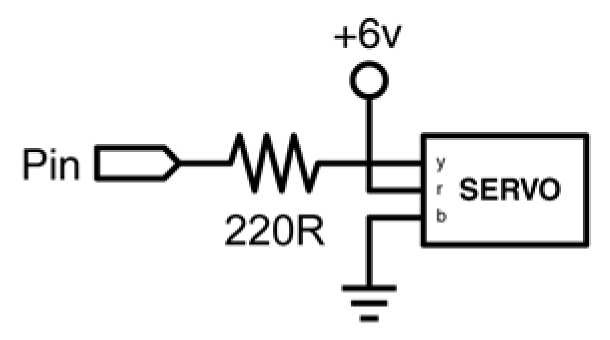
Las resistencias variables como los sensores de luz LCD los termistores, sensores de esfuerzos, etc, se conectan a las entradas analógicas para recoger valores de parámetros físicos. Este ejemplo hace uso de una función para leer el valor analógico y establecer un tiempo de retardo. Este tiempo controla el brillo de un diodo LED conectado en la salida.

```
int ledPin =     9;  // Salida analógica PWM  para conectar a LED
int analogPin =0;  // resistencia variable conectada a la entrada analógica pin 0

void setup(){}   // no  es necesario configurar entradas y salidas

void loop()
{
        for (int i=0; i<=255; i++) // incremento de valor de i
        {
        analogWrite(ledPin, i); // configura el nivel brillo con el valor de i
        delay(delayVal()); // espera un tiempo
        }
        for (int i=255; i>=0; i--) // decrementa el valor de i
        {
        analogWrite(ledPin, i); // configura el nivel de brillo con el valor de i
        delay(delayVal()); // espera un tiempo
        }
}
int delayVal() // Método para recoger el tiempo de retardo
{
        int v; // crea una variable temporal (local)
        v     = analogRead(analogPin); // lee valor analógico
        v /= 8;     // convierte el valor leído de 0-1024 a 0-128
        return v; // devuelve el valor v
}
```

salida conectada a servo



Los servos de los juguetes tienen un tipo de motor que se puede mover en un arco de 180 º y contienen la electrónica necesaria para ello. Todo lo que se necesita es un pulso enviado cada 20ms. Este ejemplo utiliza la función servoPulse para mover el servo de 10º a 170 º.

```
int servoPin = 2; // servo conectado al pin digital 2
int myAngle; // ángulo del servo de 0-180
int pulseWidth; // anchura del pulso para la función servoPulse

void setup()
{
      pinMode(servoPin, OUTPUT); // configura pin 2 como salida
}
void servoPulse(int servoPin, int myAngle)
{
      pulseWidth = (myAngle * 10) + 600; // determina retardo
      digitalWrite(servoPin, HIGH); // activa el servo
      delayMicroseconds(pulseWidth); // pausa
      digitalWrite(servoPin, LOW); // desactiva el servo
      delay(20); // retardo de refresco
}
void loop()
{
// el servo inicia su recorrido en 10º y gira hasta 170º
for (myAngle=10; myAngle<=170; myAngle++)
      {
            servoPulse(servoPin, myAngle);
      }
// el servo vuelve desde 170º hasta 10º
for (myAngle=170; myAngle>=10; myAngle--)
      {
            servoPulse(servoPin, myAngle);
      }
}
```

Como escribir una librería para Arduino

Este documento explica cómo crear una librería para Arduino. Se comienza con un programa que realiza, mediante encendido y apagado de un led, el código morse y se explica cómo convertir este en una función de librería. Esto permite a otras personas utilizar fácilmente el código que has escrito cargándolo de una forma sencilla.

Se comienza con el programa de un sencillo código Morse:

La palabra a generar es SOS (. . .   - - -   . . .)

```
// Genera SOS en código Morse luminoso

int pin = 13;

void setup()
{
 pinMode(pin, OUTPUT);
}

void loop()  //Programa principal que gerera “. . . “- - -“ y “. . . “
{
 dot(); dot(); dot(); //Genera la S (. . . )
 dash(); dash(); dash(); // Genera la O (- - -)
 dot(); dot(); dot(); // Genera la S (. . . )
 delay(3000); //Espera un tiempo
}

void dot()   //Procedimiento para generar un punto
{
 digitalWrite(pin, HIGH);
 delay(250);
 digitalWrite(pin, LOW);
 delay(250);
}

void dash()           //Procedimiento para generar una raya
{
 digitalWrite(pin, HIGH);
 delay(1000);
 digitalWrite(pin, LOW);
 delay(250);
}
```

Si se ejecuta este programa, se ejecuta el código SOS (llamada de solicitud de auxilio) en la salida PIN13.

El programa tiene distintas partes que tendremos que poner en nuestra librería. En primer lugar, por supuesto, tenemos las funciones ***dot()*** (punto) y ***dash()*** (raya) que se encargar de que el LED parpadeé de manera corta o larga respectivamente. En segundo lugar, tenemos la instrucción *ledPin* que utilizamos para determinar el pin a utilizar. Por último, está la llamada a la función ***pinMode ()*** que inicializa el pin como salida.

Vamos a empezar a convertir el programa en una librería.

Usted necesita por lo menos dos archivos en una librería: un archivo de cabecera (w / la extensión. H) y el archivo fuente (w / extensión. CPP). El fichero de cabecera tiene definiciones para la librería: básicamente una lista de todo lo que contiene, mientras que el archivo fuente tiene el código real. Vamos a llamar a nuestra biblioteca "Morse", por lo que nuestro fichero de cabecera se **Morse.h**. Echemos un vistazo a lo que sucede en ella. Puede parecer un poco extraño al principio, pero lo entenderá una vez que vea el archivo de origen que va con ella.

El núcleo del archivo de cabecera consiste en una línea para cada función en la biblioteca, envuelto en una clase junto con las variables que usted necesita:

```
class Morse
{
  public:
    Morse(int pin);
    void dot();
    void dash();
  private:
    int _pin;
};
```

Una clase es simplemente una colección de funciones y variables que se mantienen unidos todos en un solo lugar. Estas funciones y variables pueden ser públicos, lo que significa que puede ser utilizadas por quienes utilizan la librería, o privadas, lo que significa que sólo se puede acceder desde dentro de la propia clase. Cada clase tiene una función especial conocida como un **constructor**, que se utiliza para crear una instancia de la clase. El constructor tiene el mismo nombre que la clase, y no devuelve nada.

Usted necesita dos cosas más en el fichero de cabecera. Uno de ellos es un **# include** declaración que le da acceso a los tipos estándar y las constantes del lenguaje de Arduino (esto se añade automáticamente en todos los programas que hacemos con Arduino, pero no a las librerías). Por lo que debemos incluirlas (poniéndolas por encima de la definición de clase dada anteriormente):

```
# include "WConstants.h"
```

Por último, se colocara delante del código la cabecera siguiente:

```
# ifndef Morse_h
```

#define Morse_h

// el estamento #include y el resto del código va aquí..

#endif

Básicamente, esto evita problemas si alguien accidentalmente pone # include en la librería dos veces.

Por último, por lo general, se pone un comentario en la parte superior de la librería con su nombre, una breve descripción de lo que hace, quien la escribió, la fecha y la licencia.

Echemos un vistazo a la cabecera completa disposición del fichero de cabecera h:

Fichero Morse.h

```
/*
  Morse.h - Library for flashing Morse code.
  Created by David A. Mellis, November 2, 2007.
  Released into the public domain.
*/
#ifndef Morse_h
#define Morse_h
#include "WConstants.h"

class Morse
{
  public:
    Morse(int pin);
    void dot();
    void dash();
  private:
    int _pin;
};

#endif
```

Ahora vamos a escribir las diversas partes del archivo fuente de la librería, **Morse.cpp**.

Primero se ponen un par de declaraciones mediante “# include”. Estas incluyen resto del código de acceso a las funciones estándar de Arduino, ya que en las definiciones figuran en el archivo de cabecera:

```
#include "WProgram.h"
#include "Morse.h"
```

Luego viene el **constructor**. Ahora se indicará lo que debería suceder cuando alguien crea una instancia a la clase. En este caso, el usuario especifica el pin que les gustaría utilizar. Configuramos el pin como salida guardarlo en una variable privada para su uso en las otras funciones:

```
Morse::Morse(int pin)
{
  pinMode(pin, OUTPUT);
  _pin = pin;
}
```

Hay un par de cosas extrañas en este código. El primero es el **Morse::** antes del nombre de la función. Esto indica que la función es parte de la clase **Morse**. Verá este de nuevo en las otras funciones en la clase. La segunda cosa inusual es el guión bajo en el nombre de nuestra variable privada, **_pin**. Esta variable puede tener cualquier nombre que desee, siempre y cuando coincida con la definición que figura en el fichero de cabecera. La adición de un guión bajo al comienzo del nombre es una convención para dejar claro que las variables son privadas, y también a distinguir el nombre de la del argumento a la función (**pin** en este caso).

Después viene el código del programa que queremos convertir en una función (¡por fin!). Parece casi igual, excepto con **Morse::** delante de los nombres de las funciones, y **_pin** en lugar de **pin**:

```
void Morse::dot()
{
  digitalWrite(_pin, HIGH);
  delay(250);
  digitalWrite(_pin, LOW);
  delay(250);
}

void Morse::dash()
{
  digitalWrite(_pin, HIGH);
  delay(1000);
  digitalWrite(_pin, LOW);
  delay(250);
}
```

Por último, es típico incluir el comentario de cabecera en la parte superior de la fuente así como el archivo. Vamos a ver el fichero completo:

Fichero Morse.cpp

```
/*
  Morse.cpp - Library for flashing Morse code.
  Created by David A. Mellis, November 2, 2007.
  Released into the public domain.
*/

#include "WProgram.h"
#include "Morse.h"

Morse::Morse(int pin)
{
   pinMode(pin, OUTPUT);
   _pin = pin;
}

void Morse::dot()
{
   digitalWrite(_pin, HIGH);
   delay(250);
   digitalWrite(_pin, LOW);
   delay(250);
}

void Morse::dash()
{
   digitalWrite(_pin, HIGH);
   delay(1000);
   digitalWrite(_pin, LOW);
   delay(250);
}
```

Y eso es todo lo que necesita (hay algunas otras cosas opcionales, pero vamos a hablar de eso más adelante).

Ahora vamos a ver cómo se utiliza la librería.

En primer lugar, debemos crear una carpeta llamada **Morse** dentro del subdirectorio **hardware/libraries** de la aplicación Arduino. Copiar o mover los archivos **Morse.h** y **Morse.cpp** en esa carpeta. Ahora lanzar la aplicación Arduino. Cuando se inicia, compilará la recién creada librería, generando un fichero objeto (**Morse.o**) y mostrando cualquier tipo de advertencias o errores. Si usted abre el menú **Sketch> Import Library**, usted deberá ver el interior el fichero objeto Morse. Como usted trabaja con su librería, tendrá que borrar el archivo Morse.o y relanzar Arduino (o elegir una nueva tarjeta en el menú **Tools>Boards**) para recompilar su biblioteca. Si la biblioteca no se

construye, asegúrese de que están realmente los archivos CPP y. H (con y sin suplemento alguno. Pde o la extensión. Txt, por ejemplo).

Veamos como podemos escribir nuestro nuevo programa SOS haciendo uso de la nueva librería:

Programa para Arduino

```
#include <Morse.h>

Morse morse(13);

void setup()
{
}

void loop()
{
  morse.dot(); morse.dot(); morse.dot();
  morse.dash(); morse.dash(); morse.dash();
  morse.dot(); morse.dot(); morse.dot();
  delay(3000);
}
```

Hay algunas diferencias con respecto al antiguo programa (además del hecho de que algunos de los códigos se han incorporado a la librería).

En primer lugar, hemos añadido un estamento **"# include"** en la parte superior del programa. Esto hace que la librería Morse quede a disposición del programa y la incluye en el código. Esto significa que ya no necesitan una librería en el programa, usted debe borrar el # include para ahorrar espacio.

En segundo lugar, nosotros ahora podemos crear una instancia de la clase Morse llamado **morse**:

Morse morse(13);

Cuando esta línea se ejecuta (que en realidad sucede antes incluso de **setup()**), el constructor de la clase Morse será invocado y le pasara el argumento que se ha dado aquí (en este caso, sólo **13**).

Tenga en cuenta que nuestra parte **setup()** del programa está vacía, porque la llamada a **pinMode ()** se lleva a cabo en el interior de la librería (cuando la instancia se construye).

Por último, para llamar a las funciones punto **dot()** y raya **dash()**, es necesario colocar el prefijo **morse**. – delante de la instancia que queremos usar. Podríamos tener varias

instancias de la clase Morse, cada uno en su propio pin almacenados en la variable privada _pin de esa instancia. Al llamar una función en un caso particular, especificaremos qué variables del ejemplo debe utilizarse durante esa llamada a una función. Es decir, si hemos escrito:

Morse morse(13);
Morse morse2(12);

entonces dentro de una llamada a **morse2.dot (), _pin** sería 12.

Si ha escrito el nuevo programa, probablemente se habrá dado cuenta de que ninguna de nuestras funciones de la librería fue reconocida por el entorno de Arduino destacando su color. Por desgracia, el software de Arduino no puede averiguar automáticamente lo que se ha definido en su librería (a pesar de que sería una característica interesante), lo que tiene que darle un poco de ayuda. Para hacer esto, cree un archivo llamado **keywords.txt** Morse en el directorio. Debe tener un aspecto como este:

```
Morse   KEYWORD1
dash    KEYWORD2
dot     KEYWORD2
```

Cada línea tiene el nombre de la palabra clave, seguida de un código (sin espacios), seguido por el tipo de palabra clave. Las clases deben ser KEYWORD1 y son de color naranja; funciones deben ser KEYWORD2 y será de color marrón. Tendrás que reiniciar el entorno Arduino para conseguir reconocer las nuevas palabras clave.

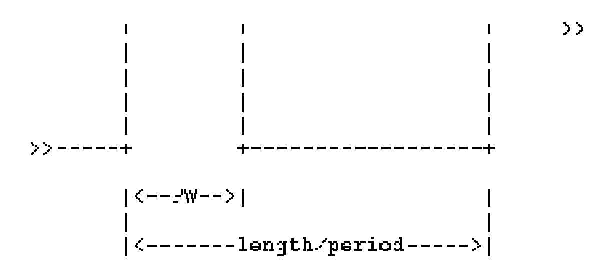
Es interesante que quienes utilicen la libreia Morse tengan algun ejemplo guardado y que aparezca en el IDE Arduino cuando seleccionamos dentro de la carpeta ejemplos (Sketch). Para hacer esto, se crea una carpeta de **ejemplos** dentro de la carpeta que contiene la librería Morse. A continuación, movemos o copiamos el directorio que contiene el programa (lo llamaremos SOS) que hemos escrito anteriormente en el directorio de ejemplos. (Usted puede encontrar el ejemplo mediante el menú **Sketch> Sketch Show Folder.)** Si reiniciamos Arduino reiniciar veremos una **Library_Morse** dentro del menú **File > Sketchbook > Examples** que contiene su ejemplo. Es posible que desee añadir algunos comentarios que explicar mejor cómo utilizar la biblioteca.

Si deseas probar la librería completa(con palabras clave y el ejemplo), puede descargarlo en: Morse.zip.

Señales analógicas de salida en Arduino (PWM).

En este apartado vamos a ver los fundamentos en los que se basa la generación de salidas analógicas en Arduino. El procedimiento para generar una señal analógica es el llamado PWM.

Señal PWM (Pulse-width modulation) señal de modulación por ancho de pulso.



Donde:

- PW (Pulse Width) o ancho de pulso, representa al ancho (en tiempo) del pulso.

- period/length (periodo), o ciclo , es el tiempo total que dura la señal.

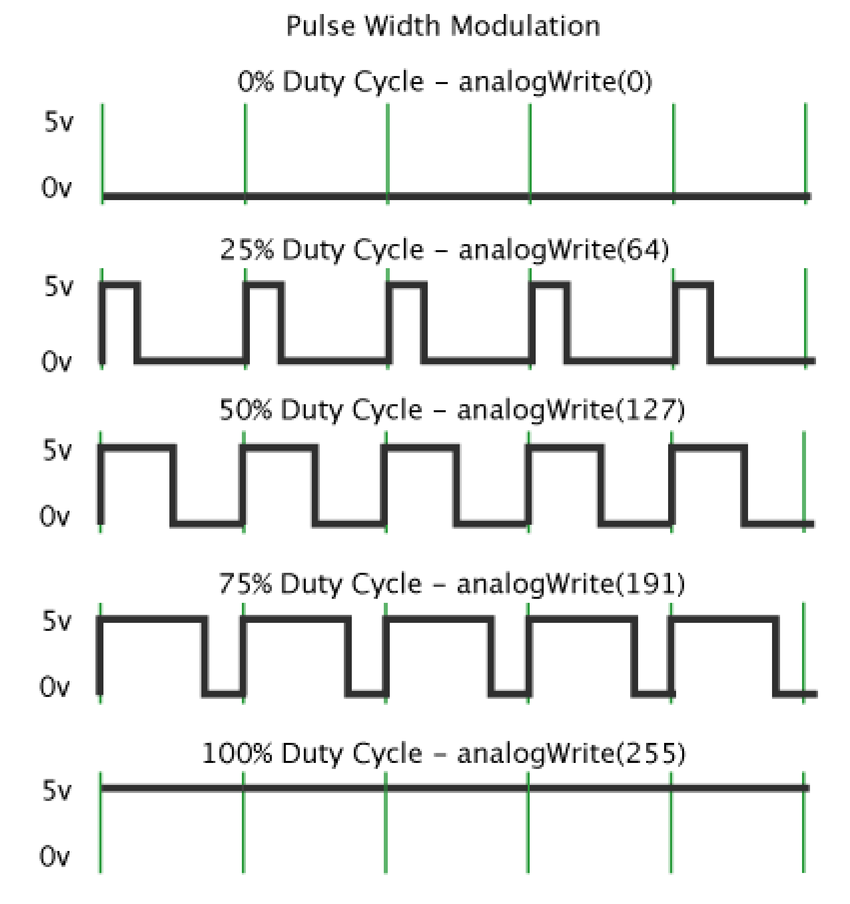
La frecuencia se define como la cantidad de pulsos (estado on/off)por segundo y su expresión matemática es la inversa del periodo, como muestra la siguiente ecuación.

$$\text{frequency} = \frac{1}{\text{period}}$$

El periodo se mide en segundos, de este modo la unidad en la cual se mide la frecuencia (hertz) es la inversa a la unidad de tiempo (segundos).

Existe otro parámetro asociado o que define a la señal PWM, denominado "Duty cycle", el cual determina el porcentaje de tiempo que el pulso (o voltaje aplicado) está en estado activo (on) durante un ciclo.

Por ejemplo, si una señal tiene un periodo de 10 ms y sus pulsos son de ancho (PW) 2ms, dicha señal tiene un duty cycle de 20% (20% on y 80% off). El siguiente gráfico muestra tres señales PWM con diferentes "duty cycles".



La señal PWM se utiliza como técnica para controlar circuitos analógicos. El periodo y la frecuencia del tren de pulsos puede determinar la potencia entregada a dicho circuito. Si, por ejemplo, tenemos un voltaje de 9v y lo modulamos con un duty cycle del 10%, obtenemos 0.9V de señal analógica de salida.

Las señales PWM son comúnmente usadas para el control de motores DC (si decrementas la frecuencia, la inercia del motor es más pequeña y el motor se mueve más lentamente), ajustar la intensidad de brillo de un LED, etc.

En Arduino la señal de salida PWM (pines 9,10) es una señal de frecuencia constante (30769 Hz) y que sólo nos permite cambiar el "duty cycle" o el tiempo que el pulso está activo (on) o inactivo (off), utilizando la función analogWrite().

Otra forma de generar señales PWM es utilizando la capacidad del microprocesador. La señal de salida obtenida de un microprocesador es una señal digital de 0 voltios (LOW) y de 5 voltios (HIGH).

Con el siguiente código y con sólo realizar modificaciones en los intervalos de tiempo que el pin seleccionado tenga valor HIGH o LOW, a través de la función digitalWrite (), generamos la señal PWM.

```
/* señal PWM */

int digPin = 10; // pin digital 10

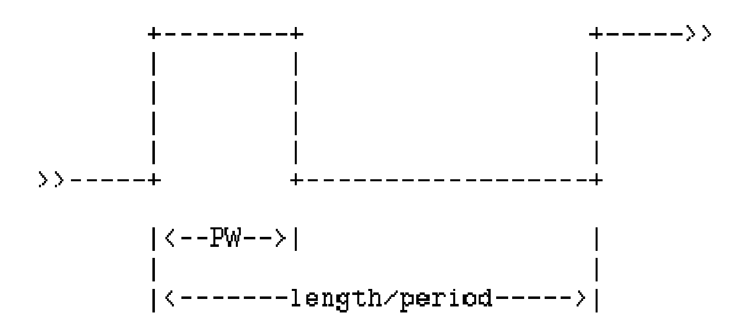
void setup() {
```

```
    pinMode(digPin, OUTPUT);   // pin en modo salida
}

void loop() {
    digitalWrite(digPin, HIGH); // asigna el valor HIGH al pin
    delay(500);                 // espera medio segundo
    digitalWrite(digPin, LOW);  // asigna el valor LOW al pin
    delay(500);                 // espera medio segundo
}
```

El programa pone el pin a HIGH una vez por segundo, la frecuencia que se genera en dicho pin es de 1 pulso por segundo o 1 Hertz de pulso de frecuencia (periodo de 1 segundo) . Cambiando la temporización del programa, podremos cambiar la frecuencia de la señal. Por ejemplo, si cambiamos las dos líneas con delay(500) a delay(250), multiplicaremos la frecuencia por dos, de forma que estamos enviando el doble de la cantidad de pulsos por segundo que antes.

Calculo de tonos:



Donde:

Frecuencia-tono=1/length-Periodo

Si "duty cycle"=50%, es decir, el ancho de los pulsos activos (on) e inactivos (off) son iguales---> Periodo=2*PW

Obteniendo la siguiente fórmula matemática:

PW o ancho de pulso = 1/(2 * toneFrequency) = period / 2

De forma que a una frecuencia o periodo dados, podemos obtener la siguiente tabla:

Nota musical	Frecuencia-tono	Periodo (us)	PW (us)
c	261 Hz	3830	1915
d	294 Hz	3400	1700
e	329 Hz	3038	1519
f	349 Hz	2864	1432
g	392 Hz	2550	1275
a	440 Hz	2272	1136
b	493 Hz	2028	1014
C	523 Hz	1912	956

(cleft) 2005 D. Cuartielles for K3

Con Arduino, tenemos dos formas de generar tonos. Con el primer ejemplo construiremos y enviaremos una señal cuadrada de salida al piezo, mientras que con el segundo haremos uso de la señal de modulación por ancho de pulso o PWM de salida en Arduino.

Ejemplo 1:

*/*Con el siguiente código y con sólo realizar modificaciones en los intervalos de tiempo que el pin seleccionado tenga valor HIGH o LOW, a través de la función*

```
digitalWrite (), generamos la señal PWM a una determinada frecuencia de
salida=261Hz*/

int digPin = 10; // pin digital 10

int PW=1915; // valor que determina el tiempo que el pulso va a estar en on/off

void setup() {
   pinMode(digPin, OUTPUT);    // pin digital en modo salida
}

void loop() {
   delayMicroseconds(PW);          // espera el valor de PW
   digitalWrite(digPin, LOW);  // asigna el valor LOW al pin
   delayMicroseconds(PW);         // espera el valor de PW
   digitalWrite(digPin, HIGH); // asigna el valor HIGH al pin
}
```

Ejemplo 2:

En Arduino la señal de salida PWM (pines 9,10) es una señal de frecuencia constante (30769 Hz) y que sólo nos permite cambiar el "duty cycle" o el tiempo que el pulso está activo (on) o inactivo (off), utilizando la función analogWrite().

Usaremos la característica "Pulse Width" con "analogWrite" para cambiar el volumen.

analogWrite(, value)

value: representa al parámetro "duty cycle" (ver PWM) y puede tomar valores entre 0 y 255.

0 corresponde a una señal de salida de valor constante de 0 v (LOW) o 0% de "duty cycle";

255 es una señal de salida de valor constante de 5 v (HIGH) o 100% de "duty cycle"; .

Para valores intermedios, el pin rápidamente alterna entre 0 y 5 voltios - el valor más alto, lo usual es que el pin esté en high (5 voltios).

La frecuencia de la señal PWM es constante y aproximadamente de 30769 Hz.

```
int speakerOut = 9; int volume = 300; // máximo volume es 1000 ¿?
 int PW=1915;
void loop() {
analogWrite(speakerOut, 0);
analogWrite(speakerOut,volume);
delayMicroseconds(PW);
analogWrite(speakerOut, 0);
delayMicroseconds(PW); }
```

Comunicando Arduino con otros sistemas

Hoy en día la manera más común de comunicación entre dispositivos electrónicos es la comunicación serial y Arduino no es la excepción. A través de este tipo de comunicación podremos enviar datos a y desde nuestro Arduino a otros microcontroladores o a un computador corriendo alguna plataforma de medios (Processing, PD, Flash, Director, VVVV, etc.). En otras palabras conectar el comportamiento del sonido o el video a sensores o actuadores. Explicaré aquí brevemente los elementos básicos de esta técnica:

Funciones básicas

El mismo cable con el que programamos el Arduino desde un computador es un cable de comunicación serial. Para que su función se extienda a la comunicación durante el tiempo de ejecución, lo primero es abrir ese puerto serial en el programa que descargamos a Arduino. Para ello utilizamos la función

beginSerial(19200);

Ya que solo necesitamos correr esta orden una vez, normalmente iría en el bloque void setup(). El número que va entre paréntesis es la velocidad de transmisión y en comunicación serial este valor es muy importante ya que todos los dispositivos que van a comunicarse deben tener la misma velocidad para poder entenderse. 19200 es un valor estándar y es el que tienen por defecto Arduino al iniciar.

Una vez abierto el puerto lo más seguro es que luego queramos enviar al computador los datos que vamos a estar leyendo de uno o varios sensores. La función que envía un dato es

Serial.print(data);

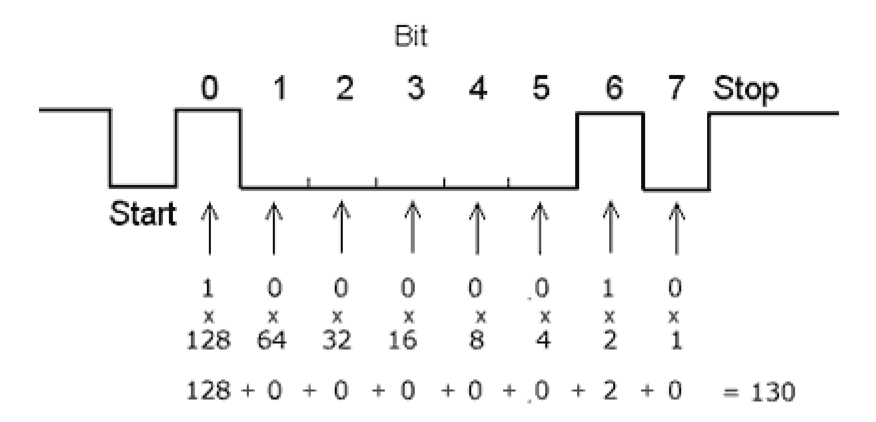
Una mirada en la referencia de Arduino permitirá constatar que las funciones print y println (lo mismo que la anterior pero con salto de renglón) tienen opcionalmente un modificador que puede ser de varios tipos:

Serial.print(data, DEC); *// decimal en ASCII*
Serial.print(data, HEX); *// hexadecimal en ASCII*
Serial.print(data, OCT); *// octal en ASCII*
Serial.print(data, BIN); *// binario en ASCII*
Serial.print(data, BYTE); *// un Byte*

Como puede verse, prácticamente todos los modificadores, menos uno, envían mensajes en ASCII. Explicaré brevemente:

Series de pulsos

En el modo más sencillo y común de comunicación serial (asincrónica, 8 bits, más un bit de parada) siempre se está enviando un byte, es decir un tren de 8 pulsos de voltaje legible por la máquina como una serie de 8, 1s ó 0s:



O sea que no importa cual modificador usemos siempre se están enviando bytes. La diferencia esta en lo que esos bytes van a representar y sólo hay dos opciones en el caso del Arduino: una serie de caracteres ASCII o un número.

Si Arduino lee en un sensor analógico un valor de 65, equivalente a la serie binaria 01000001 esta será enviada, según el modificador, como:

dato	Modificador	Envío (pulsos)
65	---DEC----	(“6” y “5” ACIIs 54–55) 000110110–000110111
65	---HEX----	(“4” Y “1” ACIIs 52–49) 000110100–000110001
65	---OCT----	(“1”, “0” y “1” ACIIs 49–48–49) 000110001–000110000–000110001
65	---BIN----	(“0”,”1”,”0”,”0”,”0”,”0”,”0”y”1” ACIIs 49–48–49–49–49–49–49–48) 000110000-…
65	---BYTE---	01000001

No explicaremos conversiones entre los diferentes sistemas de representación numérica, ni la tabla ASCII (google), pero es evidente como el modificador BYTE permite el envío de información más económica (menos pulsos para la misma cantidad de información) lo que implica mayor velocidad en la comunicación. Y ya que esto es importante cuando se piensa en interacción en tiempo real es el modo que usaremos acá.

Un ejemplo sencillo

Enviar un sólo dato es realmente fácil. En el típico caso de un potenciómetro conectado al pin 24 del ATmega:

```
int potPin = 2;
int ledPin = 13;
int val = 0;

void setup() {
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
  digitalWrite(ledPin, HIGH);  //activamos  el pin para saber cuando arranco
}

void loop() {
```

```
  val = analogRead(potPin);    // lee el valor del Pot
  Serial.println(val);
}
```

Si no utilizamos ningún modificador para el Serial.println es lo mismo que si utilizáramos el modificador DEC. Así que no estamos utilizando el modo más eficiente pero si el más fácil de leer en el mismo Arduino. Al correr este programa podremos inmediatamente abrir el monitor serial del software Arduino (último botón a la derecha) y aparecerá el dato leído en el potenciómetro tal como si usáramos el println en Processing.

Envío a Processing (versión ultra simple)

Para enviar este mismo dato a Processing si nos interesa utilizar el modo BYTE así que el programa en Arduino quedaría así:

```
int potPin = 2;
int ledPin = 13;
int val = 0;

void setup() {
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
  digitalWrite(ledPin, HIGH);  // activamos el pin para saber cuando arranco
}

void loop() {
;   // lee el Pot y lo divide entre 4 para quedar entre 0-255
  val = analogRead(potPin)/4
  Serial.print(val, BYTE);
}
```

En Processing tenemos que crear un código que lea este dato y haga algo con él:

```
import processing.serial.*;

Serial puerto;// Variable para el puerto serial
byte pot;// valor entrante
int PosX;

void setup() {
  size(400, 256);
  println(Serial.list()); // lista los puertos seriales disponibles
//abre el primero de esa lista con velocidad 9600
```

```
  port = new Serial(this, Serial.list()[0], 9600);
  fill(255,255,0);
  PosX = 0;
  pot = 0;
}

void draw() {
  if (puerto.available() > 0) { // si hay algún dato disponible en el puerto
    pot = puerto.read();// lo obtiene
    println(pot);
  }
  ellipse(PosX, pot, 3, 3); // y lo usa
  if (PosX < width) {
    PosX++;
  } else {
    fill(int(random(255)),int(random(255)),int(random(255)));
    PosX = 0;
  }
}
```

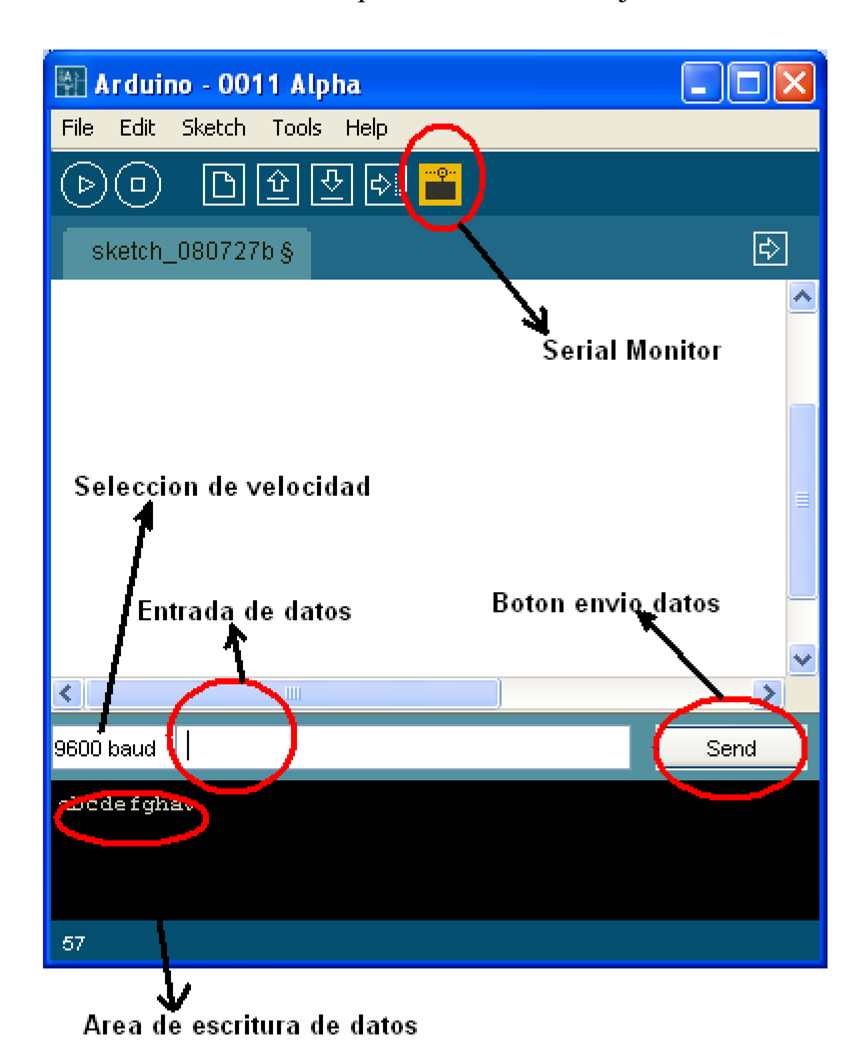
Si ya se animó a intentar usar más de un sensor notará que no es tan fácil como duplicar algunas líneas.

Comunicación vía puerto Serie:

La tarjeta Arduino puede establecer comunicación serie (recibir y enviar valores codificados en ASCII) con un dispositivo externo, a través de una conexión por un cable/puerto USB (tarjeta USB) o cable/puerto serie RS-232(tarjeta serie) (Enlace)

Igual que para la descarga de los programas, sólo será necesario indicar el número de puerto de comunicaciones que estamos utilizando y la velocidad de transferencia en baudios (enlace).También hay que tener en cuenta las limitaciones de la transmisión en la comunicación serie, que sólo se realiza a través de valores con una longitud de 8-bits (1 Byte)(Ver serialWrite(c) o serialRead(c)), mientras que como ya se hemos indicado, el A/D (Convertidor) de Arduino tiene una resolución de 10-bits.(enlace)

Dentro del interfaz Arduino, disponemos de la opción "Monitorización de Puerto Serie", que posibilita la visualización de datos procedentes de la tarjeta.



Para definir la velocidad de transferencia de datos, hay que ir al menú "Herramientas" y seleccionar la etiqueta "Velocidad de monitor Serie". La velocidad seleccionada, debe coincidir con el valor que hemos determinado o definido en nuestro programa y a través

del comando beginSerial().Dicha velocidad es independiente de la velocidad definida para la descarga de los programas.

La opción de "Monitorización de puerto serie" dentro del entorno Arduino, sólo admite datos procedentes de la tarjeta. Si queremos enviar datos a la tarjeta, tendremos que utilizar otros programas de monitorización de datos de puerto serie como HyperTerminal (para Windows) -Enlace o ZTerm (para Mac)-XXXX- Linux-Enlace, etc.

También se pueden utilizar otros programas para enviar y recibir valores ASCII o establecer una comunicación con Arduino: Processing (enlace), Pure Data (enlace), Director(enlace), la combinación o paquete serial proxy + Flash (enlace), MaxMSP (enlace), etc.

Nota: Hay que dejar tiempos de espera entre los envíos de datos para ambos sentidos, ya que se puede saturar o colapsar la transmisión. ¿?

Envío de datos desde Arduino(Arduino->PC) al PC por puerto de comunicación serie:

Ejercicio de volcado de medidas o valores obtenidos de un sensor analógico

Código

```
/* Lectura de una entrada analógica en el PC
 El programa lee una entrada analógica, la divide por 4
 para convertirla en un rango entre 0 y 255, y envía el valor al PC en
 diferentes formatos ASCCI.
 A0/PC5: potenciómetro conectado al pin analógico 1 y puerto de PC-5
 Created by Tom Igoe 6 Oct. 2005
 Updated
 */
int val; // variable para capturar el valor del sensor analógico

void setup() {
  // define la velocidad de transferencia a 9600 bps (baudios)
  beginSerial(9600);
}

void loop() {
  // captura la entrada analógica, la divide por 4 para hacer el rango de 0-255
  val = analogRead(0)/4;
// texto de cabecera para separar cada lectura:
    printString("Valor Analogico =");
```

```
// obtenemos un valor codificado en ASCII (1 Byte) en formato decimal :
   printInteger(val);
   printString("\t");  //Carácter espacio
// obtenemos un valor codificado en ASCII (1 Byte) en formato hexadecimal :
   printHex(val);
   printString("\t");
// obtenemos un valor codificado en ASCII (1 Byte) en formato binario
  printBinary(val);
  printString("\t");
// obtenemos un valor codificado en ASCII (1 Byte)en formato octal:
   printOctal(val);
   printString("\n\r"); //caracter salto de linea y retorno de carro
// espera 10ms para la próxima lectura
  delay(10);
}
```

Otra solución puede ser la de transformar los valores capturados en un rango entre 0 y 9 y en modo de codificación ASCII o en caracteres ASCII. De forma que dispongamos de un formato más sencillo o legible, sobre la información capturada.

El siguiente código incluye una función llamada **treatValue()** que realiza dicha transformación.

```
int val; // variable para capturar el valor del sensor analógico

void setup() {
 // define la velocidad de transferencia a 9600 bps (baudios)
 beginSerial(9600);
}

int treatValue(int data) {
 return (data * 9 / 1024) + 48; //fórmula de transformación
 }
void loop() {

val= analogRead(0); //captura del valor de sensor analógico (0-1023)

serialWrite(treatValue(val)); //volcado al puerto serie de 8-bits
```

```
serialWrite(10); //caracter de retorno de carro

serialWrite(13); //caracter de salto de línea

delay(10); }
```

```
// Serial Output // by BARRAGAN <http://people.interaction-ivrea.it/h.barragan>

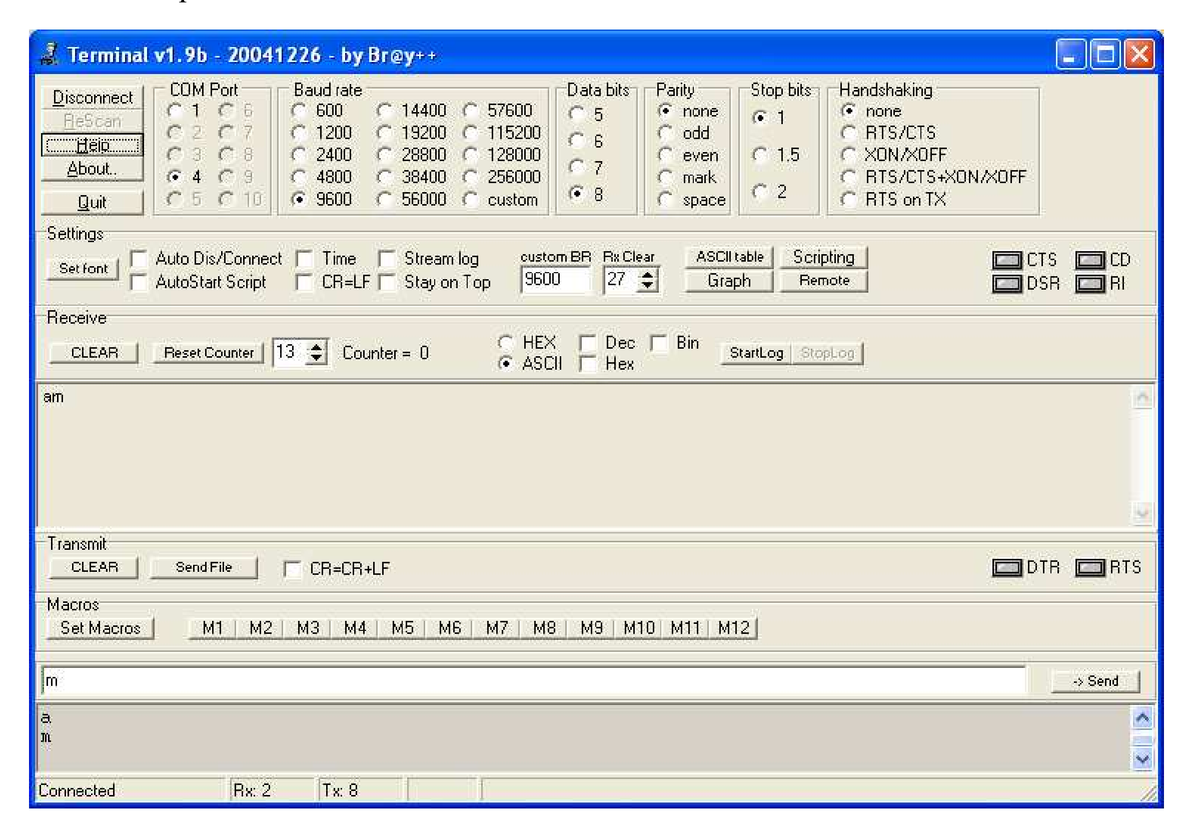
int switchpin = 0; // interruptor conectado al pin 0

void setup() {
  pinMode(switchpin, INPUT);  // pin 0 como ENTRADA
  Serial.begin(9600);       // inicia el puerto serie a 9600bps
}

void loop() {
  if(digitalRead(switchpin) == HIGH)  //si el interruptor esta en ON
  {
    Serial.print(1);        // envía 1 a Processing
  }else{
    Serial.print(0);       // en caso contrario envía 0 a Processing
  }
  delay(100);             // espera 100ms
}
```

Envío de datos desde el PC (PC->Arduino) a Arduino por puerto de comunicación serie:

En primer lugar, necesitamos instalar un programa como Hyperterminal en nuestro PC, en caso de que sea Windows........



Software Terminal para realizar comunicaciones con el puerto serie

Seleccionar el puerto que estamos utilizando con la tarjeta, la velocidad de transferencia y el formato de salida de los datos. Y finalmente conectar...

Se puede realizar una comprobación con el ejercicio mostrado arriba.

Nota: El programa de monitorización de datos está ocupando el puerto utilizado para la conexión a la tarjeta, por lo que si quieres realizar una nueva descarga del programa, tendrás que desconectarte previamente de este último.

```
/*by BARRAGAN <http://people.interaction-ivrea.it/h.barragan>
*Demuestra como leer un dato del puerto serie. Si el dato recibido es una 'H', la luz se
*enciende ON, si es una  'L', la luz se apaga OFF. Los datos provienen del PC o de un
*programa como Processing..
*created 13 May 2004 revised 28 Aug 2005
*/

char val; // variable que recibe el dato del puerto serie
int ledpin = 13; // LED conectado al pin 13
```

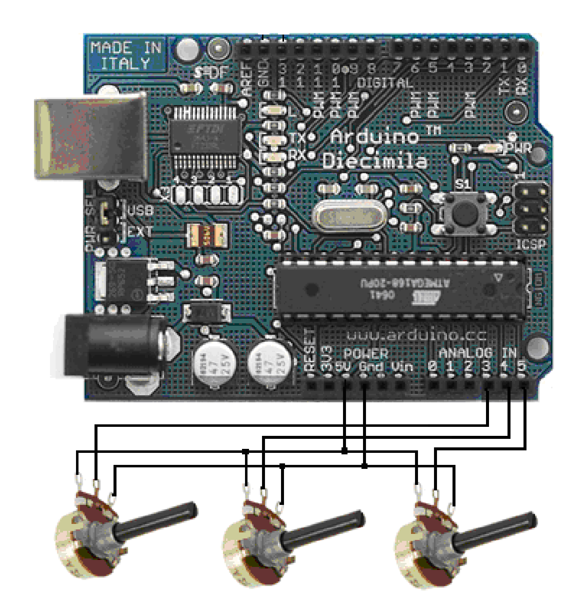
```
void setup() {
  pinMode(ledpin, OUTPUT);  // pin 13 (LED)actua como SALIDA
  Serial.begin(9600);       // inicia la comunicación con el puerto serie a 9600bps
}
void loop() {
  if( Serial.available() )       // si hay dato e el puerto lo lee
  {
    val = Serial.read();         // lee y almacena el dato en  'val'
  }
  if( val == 'H' )             //su el dato recibido es ´H´
  {
    digitalWrite(ledpin, HIGH);  //activa el LED
  } else {
    digitalWrite(ledpin, LOW);   // en caso contrario lo desactiva
  }
  delay(100);                  // espera  100ms para una nueva lectura
}
```

Para probar este programa bastará con iniciar el programa que actúe de “terminal de comunicación” Hyperterminal de Windowws o el programa mostrado anteriormente y podemos enviar los datos y comprobar como actúa.

Envío a petición (toma y dame)

Cuando se envía más de un dato del Arduino a otro sistema es necesario implementar reglas de comunicación adicionales para poder distinguir a que dato corresponde cada uno de los paquetes de bytes recibidos. Una manera simple y eficiente de hacer esto es jugando al *"toma y dame"*. Arduino no enviará los valores de los sensores hasta que Processing no le envíe también un valor por el puerto serial y Processing, a su vez, no enviara ese valor hasta no tener los datos que espera completos.

Este sería el código para Arduino usando tres potenciómetros en los últimos tres pines analógicos del ATmega:



Codigo para cargar en la tarjeta Arduino desde el IDE Arduino

```
int pot1= 0;           // valores de los sensores analógicos
int pot2= 0;
int pot3= 0;
int inByte = 0;        // valor entrante de Processing

void setup()
{
 Serial.begin(9600);
}

void loop()
{
 if (Serial.available() > 0) {  // sólo si algo ha llegado
   inByte = Serial.read();      // lo lee
   // hace la lectura de los sensores en pines 3,4y5 (análogos)
   pot1 = analogRead(3)/4;
   pot2 = analogRead(4)/4;
   pot3 = analogRead(5)/4;
```

```
    // y los envía
    Serial.print(pot1, BYTE);
    Serial.print(pot2, BYTE);
    Serial.print(pot3, BYTE);
  }
}
```

Una vez cargado este programa en la tarjeta Arduino está en disposición de enviar los datos de las lecturas de los potenciómetros cuando le sean demandados por el programa que los requiera. En nuestro ejemplo vamos a escribir un programa en el IDE Processing y será este el que se ocupe de leer los datos y con ellos modificar la posición de una bola que aparecerá en pantalla

Será processing quién empezará el "toma y dame" y deberá reconocer cada dato. Este es el código:

Código para Processing

```
import processing.serial.*;

Serial puerto;
int[] datosEntrantes = new int[3];   // arreglo para recibir los tres datos
int cuantosDatos = 0;               // contador
int posX, posY, posZ;               // posición de un objeto 3D
boolean hayDatos = false;           // control de verdad

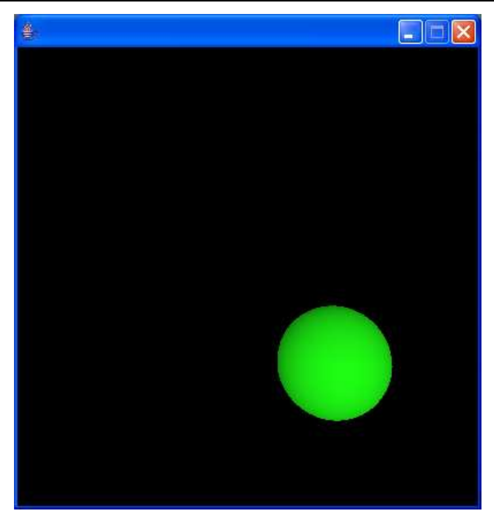
void setup() {
  size(400, 400, P3D);
  noStroke();

  println(Serial.list());// puertos serie disponibles
  puerto = new Serial(this, Serial.list()[0], 9600); // Configuración del puerto
  puerto.write(65);    // Envía el primer dato para iniciar el toma y dame
}

void draw() {
  background(0);
  lights();
  fill(30,255,20);
  translate(width/2 + posX, height/2 + posY, posZ);
  sphere(40);

  if (hayDatos == false) { //si no hay datos envía uno
    puerto.write(65);
  }
}
// esta función corre cada vez que llega un dato serial
```

```
void serialEvent(Serial puerto) {
  if (hayDatos == false) {
    hayDatos = true;    // de ahora en adelante el dato de envío se dará por el toma y
dame
  }
  // Lee el dato y lo añade al arreglo en su última casilla
  datosEntrantes[cuantosDatos] = puerto.read();
  cuantosDatos++;
  if (cuantosDatos > 2 ) { // Si ya hay tres datos en el arreglo
    posX = datosEntrantes[0];
    posY = datosEntrantes[1];
    posZ = datosEntrantes[2];
    println("Valores de los potenciometros: " + posX + "," + posY + "," + posZ);
    puerto.write(65); // y envía para pedir más
    cuantosDatos = 0; // y todo empieza de nuevo
  }
}
```



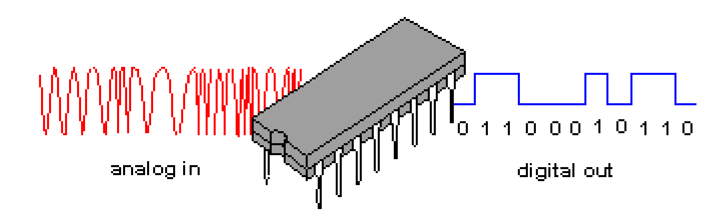
```
Valores de los potenciometros: 61,61,60
Valores de los potenciometros: 61,61,61
Valores de los potenciometros: 61,61,61
```

14

Aspecto del IDE Processing cuando esta en funcionamiento el programa de captura de valores de los tres potenciómetros.

Conversor Analógico-Digital (A/D)

Un conversor analógico-digital es un dispositivo electrónico capaz de convertir una señal analógica en un valor binario, en otras palabras, este se encarga de transformar señales analógicas a digitales (0's y 1's).



El dispositivo establece una relación entre su entrada (señal analógica) y su salida (Digital) dependiendo de su resolución . La resolución determina la precisión con la que se reproduce la señal original.

Esta resolución se pude saber, siempre y cuando conozcamos el valor máximo de la entrada a convertir y la cantidad máxima de la salida en dígitos binarios.

Resolución = +Vref/2^n(n-bits)

Por ejemplo, un conversor A/D de 8-bits puede convertir valores que van desde 0V hasta el voltaje de referencia (Vref) y su resolución será de:

Resolución = Vref/256 (2^8)

Lo que quiere decir que mapeará los valores de voltaje de entrada, entre 0 y Vref voltios, a valores enteros comprendidos entre 0 y 255 (2^{n-1}).

La tarjeta Arduino utiliza un conversor A/D de 10-bits, así que:

Resolución = Vref/1024 (2^{10})

Mapeará los valores de voltaje de entrada, entre 0 y Vref voltios, a valores enteros comprendidos entre 0 y 1023 (2^{n-1}). Con otras palabras, esto quiere decir que nuestros sensores analógicos están caracterizados con un valor comprendido entre 0 y 1023. (Ver analogRead()).

Si Vref es igual a 5v, la resolución es aproximadamente de 5 milivoltios. Por lo tanto el error en las medidas de voltaje será siempre de sólo 5 milivoltios.

Caso de transmisión o envío de datos (comunicación) por el puerto serie:

Al enviar datos por el puerto serie, tenemos que tener en cuenta que la comunicación se realiza a través de valores con una longitud de 8-bits (Ver serialWrite(c) o serialRead(c)), mientras que como ya se hemos indicado, el A/D (Convertidor) de Arduino tiene una resolución de 10-bits.

Por ejemplo, si capturamos los valores de un sensor analógico (e.j. potenciómetro) y los enviamos por el puerto serie al PC, una solución podría se transformarlos en un rango entre 0 y 9 y en modo de codificación ASCII (carácter).

(dato capturado del sensor analógico * 9 / 1024) + 48;

0 ASCII -->decimal = 48

1 ASCII -->decimal = 49

etc..

En forma de código podría quedar como:

```
value1 = analogRead(analogPin1); //captura del valor de sensor analógico (0-1023)
serialWrite(treatValue(value1)); //volcado al puerto serie 8-bits
int treatValue(int data) {
return (data * 9 / 1024) + 48;// fórmula de transformación
}
```

Otra fórmula sería dividiendo por 4 ¿Esto es correcto? (1024/256)los valores capturados de los sensores analógicos, para convertirlos en valor de byte válido (0 - 255).

```
value = analogRead(analogPin)/4;
serialWrite(value);
```

Comunicación serie

Para hacer que dos dispositivos se comuniquen necesitamos un método de comunicación y un lenguaje o protocolo común entre ambos dispositivos. La forma más común de establecer dicha comunicación es utilizando la comunicación serie. La comunicación serie consiste en la transmisión y recepción de pulsos digitales, a una misma velocidad.

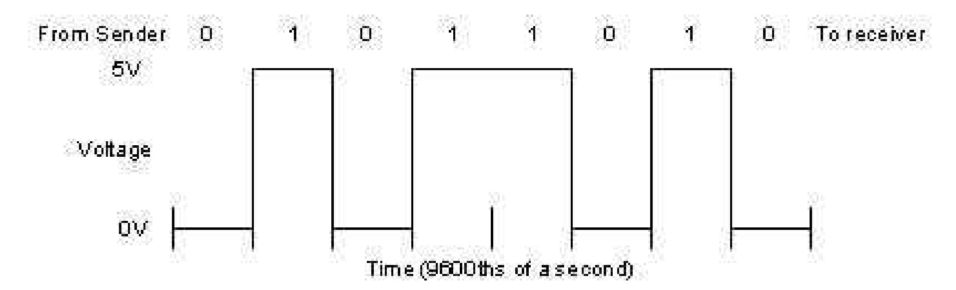
El transmisor envía pulsos que representan el dato enviado a una velocidad determinada, y el receptor escucha dichos pulsos a esa misma velocidad. Esta técnica es conocida como comunicación serie asíncrona. Un caso práctico es el de un MODEM externo conectado a un PC.

Por ejemplo, si tenemos dos dispositivos conectados y que intercambian datos a una velocidad de 9600 bits por segundo (también llamados baudios), el receptor capturará el voltaje que le está enviando el transmisor, y cada 1/9600 de un segundo, interpretará dicho voltaje como un nuevo bit de datos. Si el voltaje tiene valor HIGH (+5v en la comunicación con Arduino), interpretará el dato como 1, y si tiene valor LOW (0v), interpretará el dato como 0. De esta forma, interpretando una secuencia de bits de datos, el receptor puede obtener el mensaje transmitido.

Los dispositivos electrónicos usan números para representar en bytes caracteres alfanuméricos (letras y números). Para ello se utiliza el código estándar llamado ASCII (enlace), el cual asigna a cada número o letra el valor de un byte comprendido entre el rango de 0 a 127 ¿?. El código ASCII es utilizado en la mayoría de los dispositivos como parte de su protocolo de comunicaciones serie.

Así que si queremos enviar el número 90 desde un dispositivo a otro. Primero, se pasa el número desde su formato decimal a su formato binario. En binario 90 es 01011010 (1 byte).

Y el dispositivo lo transmitiría como secuencia de pulsos según el siguiente gráfico:



Otro punto importante, es determinar el orden de envío de los bits. Normalmente, el transmisor envía en primer lugar, el bit con más peso (o más significativo), y por último el de menos peso (o menos significativo) del formato binario.

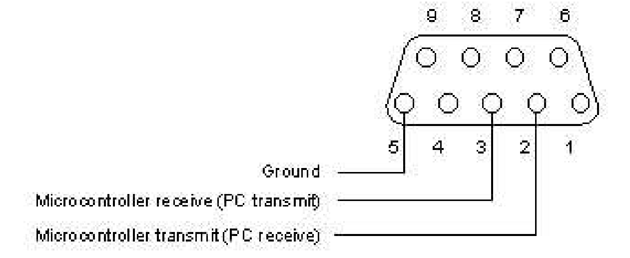
Entonces y como conclusión, para que sea posible la comunicación serie, ambos dispositivos deben concordar en los niveles de voltaje (HIGH y LOW), en la velocidad de transmisión, y en la interpretación de los bits transmitidos. Es decir, que deben de tener el mismo protocolo de comunicación serie(conjunto de reglas que controlan la

secuencia de mensajes que ocurren durante una comunicación entre dispositivos). Generalmente se usa el protocolo serie llamado RS-232 y interfaces (conectores vs puertos serie) que utilizan dicha norma.

Hasta no hace mucho, la mayoría de los PCs utilizaban el estandar RS-232 para la comunicación serie, pero actualmente los PCs están migrando hacia otras formas de comunicación serie, tales como USB (Bus Serie Universal), y Firewire, que permiten una configuración más flexible y velocidades de transmisión más altas.

Para conectar un dispositivo a un PC (o sistema operativo) necesitamos seleccionar un puerto serie y el cable apropiado para conectar al dispositivo serie.

Gráfico de Puerto serie RS-232 en PC (versión de 9 pines DB-9)



En Arduino y en función del modelo de placa que hayamos adquirido tendremos que elegir un cable RS-232 (estándar, no debe ser de tipo null modem) o USB o bien un adaptador RS-232/USB. (enlace a guía de instalación)

Palabras reservadas del IDE de Arduino

Estas palabras son constante, variables y funciones que se definen en el lenguaje de programación de Arduino. No se deben usar estas palabras clave para nombres de variables.

# Constantes	private	loop
	protected	max
HIGH	public	millis
LOW	return	min
INPUT	short	-
OUTPUT	signed	%
SERIAL	static	/*
DISPLAY	switch	*
PI	throw	new
HALF_PI	try	null
TWO_PI	unsigned	()
LSBFIRST	void	PI
MSBFIRST		return
CHANGE	**# Other**	>>
FALLING		;
RISING	abs	Serial
false	acos	Setup
true	+=	sin
null	+	sq
	[]	sqrt
# Variables de	asin	-=
designacion de puertos	=	switch
y constantes	atan	tan
	atan2	this
DDRB	&	true
PINB	|	TWO_PI
PORTB	boolean	void
PB0	byte	while
PB1	case	Serial
PB2	ceil	begin
PB3	char	read
PB4	char	print
PB5	class	write
PB6	,	println
PB7	//	available
	?:	digitalWrite
DDRC	constrain	digitalRead
PINC	cos	pinMode
PORTC	{}	analogRead
PC0	--	analogWrite
PC1	default	attachInterrupts
PC2	delay	detachInterrupts

PC3
PC4
PC5
PC6
PC7

DDRD
PIND
PORTD
PD0
PD1
PD2
PD3
PD4
PD5
PD6
PD7

Tipos de datos

boolean
byte
char
class
default
do
double
int
long

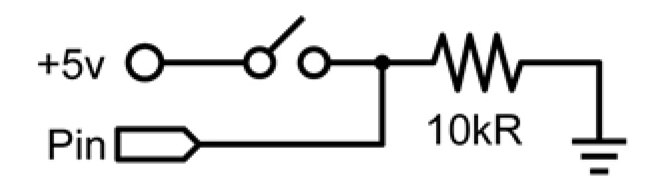
delayMicroseconds
/
/**
.
else
==
exp
false
float
float
floor
for
<
<=
HALF_PI
if
++
!=
int
<<
<
<=
log
&&
!
||

beginSerial
serialWrite
serialRead
serialAvailable
printString
printInteger
printByte
printHex
printOctal
printBinary
printNewline
pulseIn
shiftOut

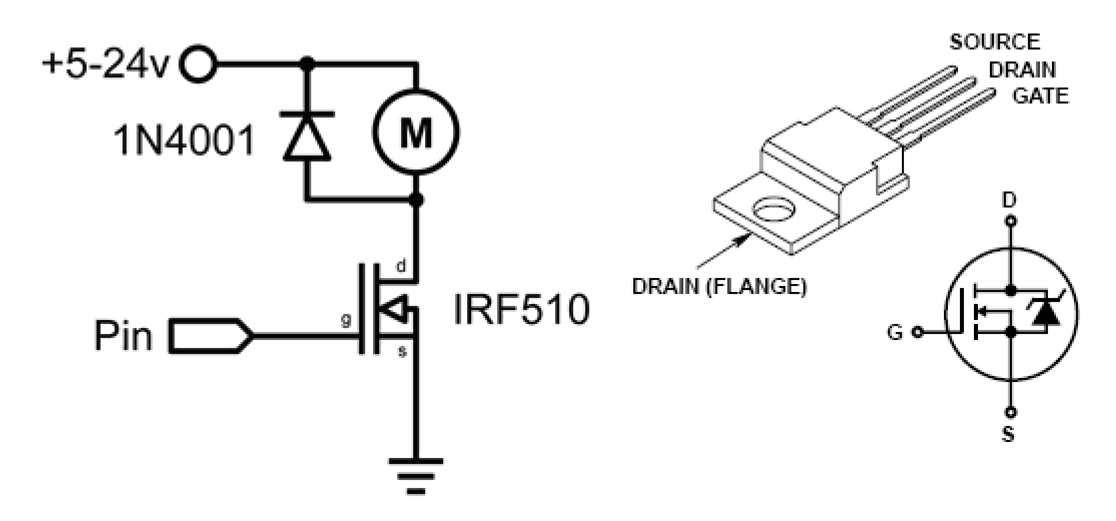
CIRCUITOS DE INTERFACE CON ARDUINO



Conexión de un diodo Led a una salida de Arduino



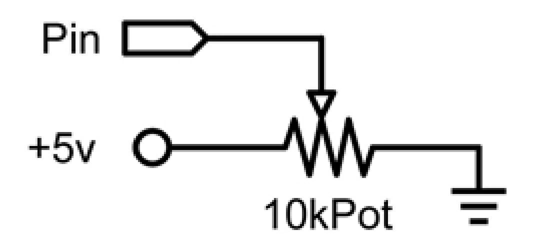
Conexión de un pulsador/interruptor



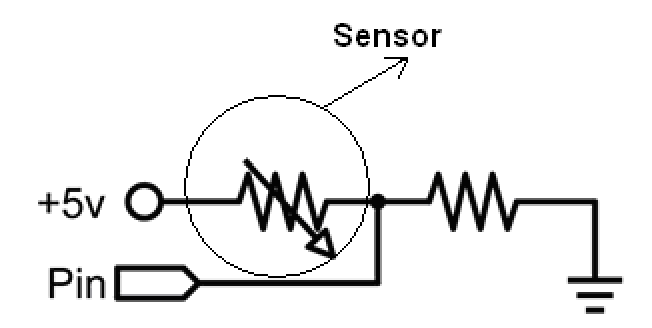
Conexión de una carga inductiva de alto consumo mediante un MOSFET



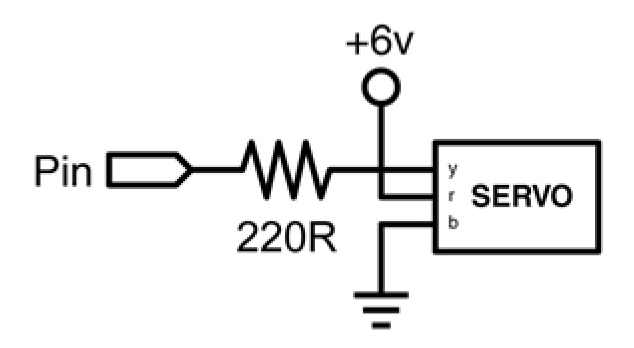
Conexión de una salida analógica a un LED



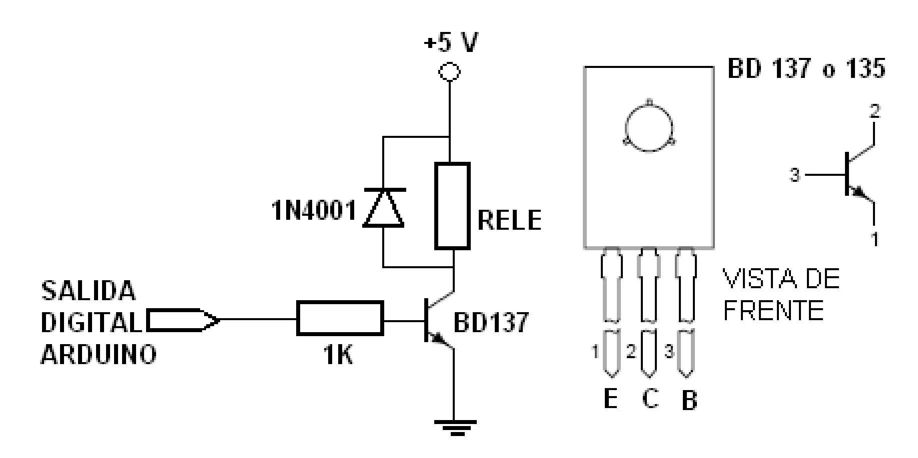
Entrada analógica mediante un potenciómetro



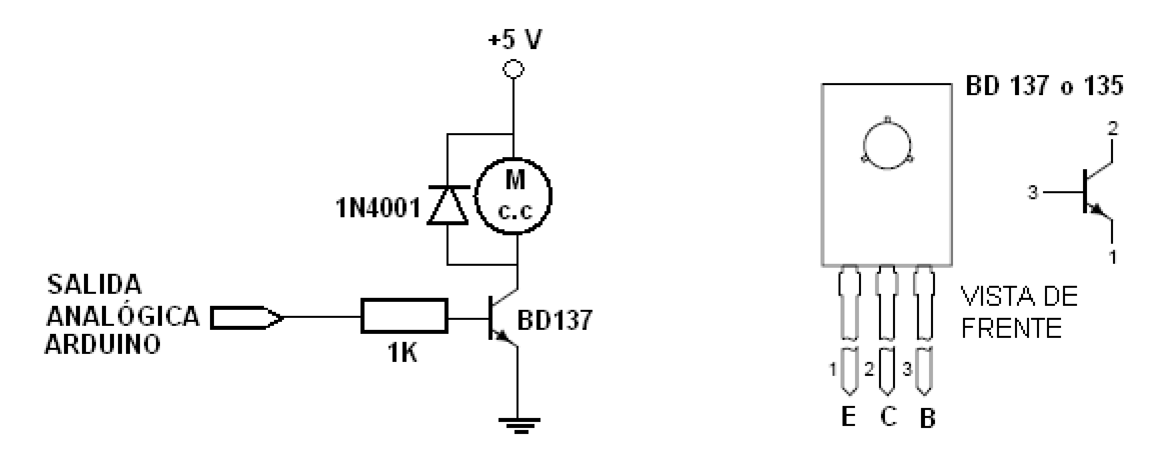
Conexión de un sensor de tipo resistivo (LRD, NTC, PTC..) a una entrada analógica



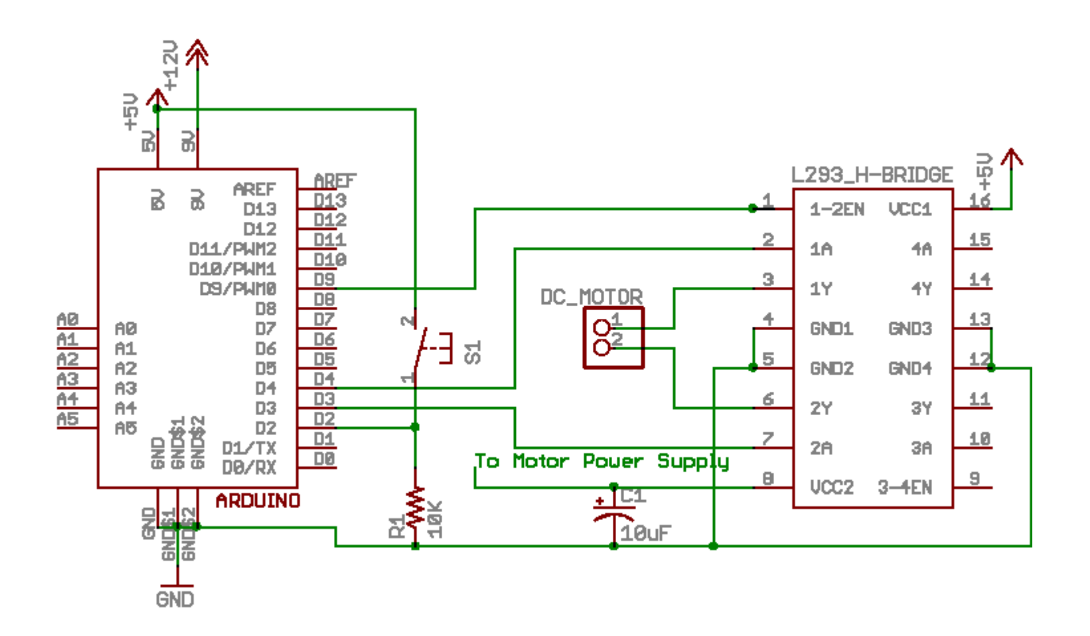
Conexión de un servo a una salida analógica.



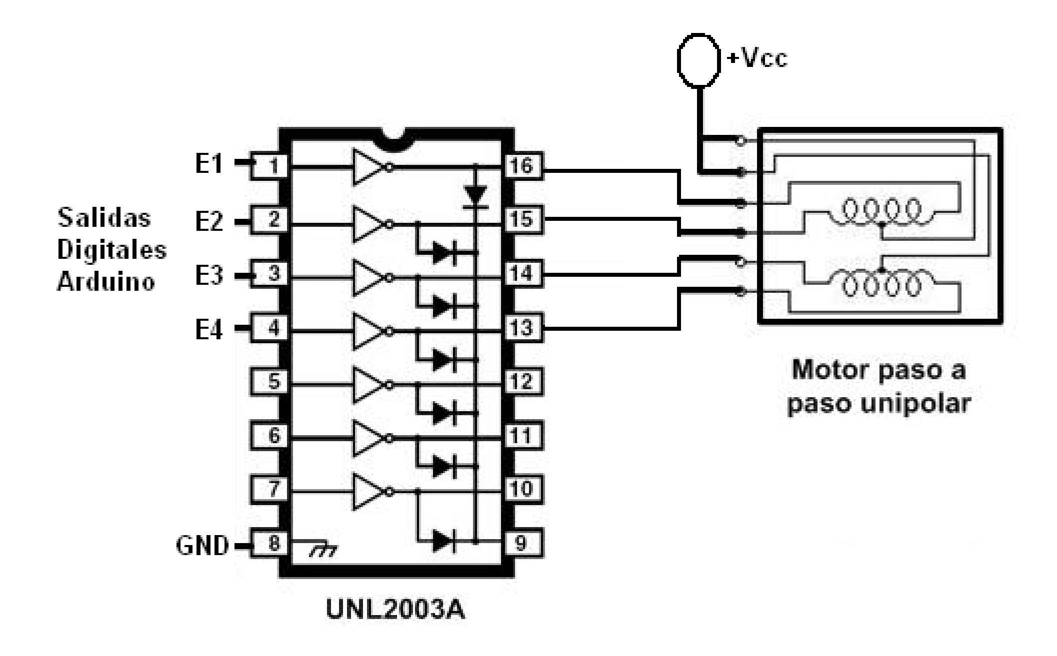
Gobierno de un Relé mediante una salida digital de Arduino



Gobierno de un motor de cc mediante una salida analógica de Arduino controlando la velocidad del motor



Control de un motor de cc mediante el CI L293



Control de un motor paso a paso unipolar

Control con transistor TIP120

Vcc (+5v)
DC_MOT
1
2
D1
1N4004
salida digital
Arduino
R2
1K
Q1
TIP120
GND
carga Inductiva

Vcc (+5v)
LMP1
salida digital
Arduino
R2
1K
Q1
TIP120
GND
cargar resistiva

Control mediante transistor TIP120